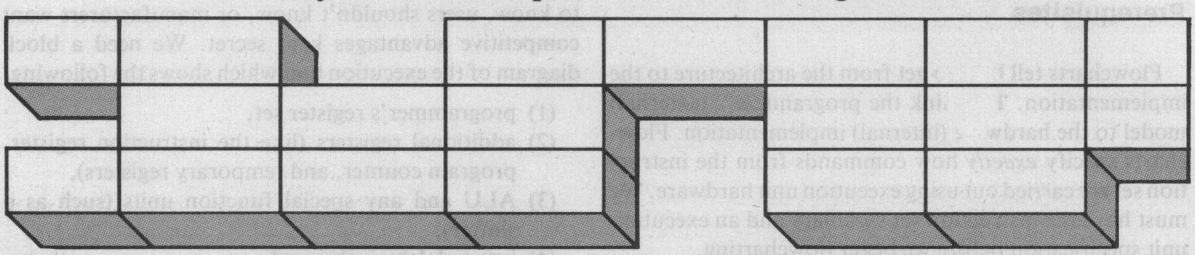
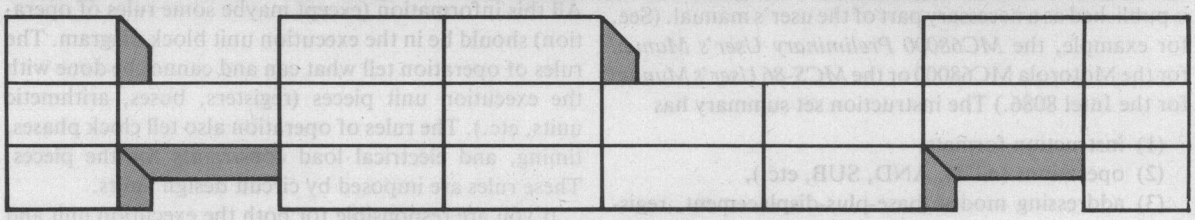
The flowchart method is a procedure for designing the processor of a computer. It helps transform the English description into the formal description a circuit designer needs.



How to Flowchart for Hardware



Nick Tredennick, IBM T. J. Watson Research Center

The "flowchart method" is a procedure for designing the processor of a computer. The technique works for big processors and little processors.

A processor has two parts: a "controller" and "execution unit." The controller tells the execution unit *what* to do *when*. The controller determines, more than anything else, the processor's "personality." The execution unit is a collection of fast but latent capabilities (registers, ALUs, shifters, and data paths) which are awakened by the controller.

Processor chip designs begin with an appeal: "We need a processor that's twice as good as any rival." "Architects" turn the petition into an English description of the machine (in IBM's System/370, this is the "Principles of Operation" manual). Engineers use the techniques of logic design and circuit design to implement the machine from the English description. We have lots of books to help us with logic design and circuit design. The trouble is, *nobody says how to transform the English description into the kind of formal description a circuit designer needs.*

It's much like a mathematical word problem. The hard part is getting the equations from the written description of the problem. Once we have the equations we can apply documented methods to find the solution. The English description of a chip is like a book-length mathematical problem! The hardware flowcharts in this paper are a bridge between English and the circuit designer. These flowcharts are a compact formal description of what the machine does.

The method I describe was used to design the controller in the MC68000 microprocessor. The flowchart method is both procedure and notation. The designer follows the procedure to express the design in the particular form I call flowcharts. Unlike most procedures, this one does not start out by *presuming* a block diagram for the machine. (Doing this *imposes* a controller structure on the English specification; the problem is to *find* an efficient structure.) The block diagram is one of the procedure's *outputs*. My flowcharts show the design as the flow of simple machine actions. An example machine action is "RX→A→ALU," which means "put the contents of register RX on the A bus to the ALU." (That also exemplifies the notation; it doesn't get more complicated than that.) One of these is called a "task"; machine states can be one or more tasks. I depict the flow of states by boxes (one for each state); I draw these in a specific format, and it matters that you draw the states precisely the way I say. With the flowchart method you see major flow (a complicated microprocessor can fit on six 8½ by 11-inch pages) without losing important detail. "RX→A→ALU" is uncluttered by the usual hardware details that hide significant controller structure issues. The hardware is debugged using the flowcharts; they are the authoritative reference for the design.

The procedure is carried out with a particular technology in mind. Decisions in the procedure *are* based on the capabilities of the particular technology. The procedure *does not* depend on the implementation method. This means the same flowcharts are used to implement the chip with combinational logic, PLAs, or microcode! I will show how to implement a simple processor using the flowcharts.

I tell how to flowchart hardware using just pencil and paper. I describe flowcharting using such simple tools because

(1) The method *is* useful whether the designer has just a desk and wastebasket, or several million dollars in computers and fancy equipment.
(2) Design automation should be subservient to the design method. It should *support* the design pro-

0018-9162/81/1200-0087$00.75 © 1981 IEEE

cedure, not *be* the design procedure. (Often, engineers' methods are solely the result of available design automation tools; I think that's bad.)

Prerequisites

Flowcharts tell l[illegible] get from the architecture to the implementation. T[illegible]ink the programmer's (external) model to the hardw[illegible] (internal) implementation. Flowcharts specify *exactly* how commands from the instruction set are carried out using execution unit hardware. We must have the instruction set summary and an execution unit specification before we begin flowcharting.

Instruction set summary. The instruction set summary is published as a necessary part of the user's manual. (See, for example, the *MC68000 Preliminary User's Manual* for the Motorola MC68000 or the *MCS-86 User's Manual* for the Intel 8086.) The instruction set summary has

(1) instruction formats,
(2) operations (ADD, AND, SUB, etc.),
(3) addressing modes (base-plus-displacement, register indirect, indexed, etc.), and
(4) registers (as seen by the programmer).

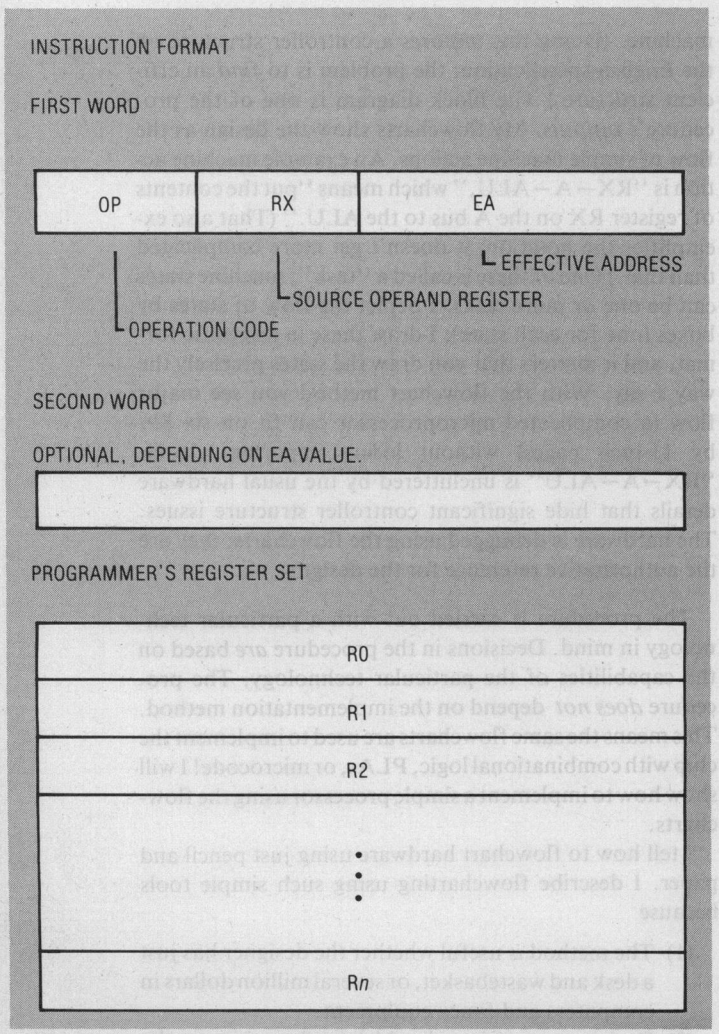


Figure 1. MIN instruction format and register set.

Execution unit. A processor's execution unit (or "data flow") details are not usually published: users don't want to know, users shouldn't know, or manufacturers want competitive advantages kept secret. We need a block diagram of the execution unit which shows the following:

(1) programmer's register set,
(2) additional registers (like the instruction register, program counter, and temporary registers),
(3) ALU and any special function units (such as a shifter),
(4) internal data paths, and
(5) rules of operation.

All this information (except maybe some rules of operation) should be in the execution unit block diagram. The rules of operation tell what can and cannot be done with the execution unit pieces (registers, buses, arithmetic units, etc.). The rules of operation also tell clock phases, timing, and electrical load constraints for the pieces. These rules are imposed by circuit design limits.

If you are responsible for both the execution unit and the flowcharts, do the execution unit first. To design the execution unit, I recommend doing trial flowcharts for 10 frequently used instructions to determine an initial execution unit structure. I think a simple bus-oriented structure is best, so I start with that. The execution unit will evolve. In a current (1981) VLSI implementation, some limits on your interconnect scheme will come from the circuit designers. For example, having no more than three buses allows bus wiring to pass right over the registers and arithmetic units without using extra chip area.

Example processor

To avoid confusing details, I illustrate the method with a simple microprocessor, called MIN. Figure 1 shows the instruction format and register set; Figure 2 shows a subset of the instruction set summary. This subset is adequate to demonstrate flowchart construction. Figure 3 shows a sufficiently detailed block diagram of the execution unit. It also includes some rules of operation; others will be added as we progress.

Figures 1, 2, and 3 don't have the usual details about word length, instuction length, address length, bus width, ALU size, and register size. Though you know this information, it doesn't necessarily change the sequence of operations for the execution unit. The sequence of operations depends on the ratio of these parameters to each other and not their absolute value. You implement the design from the flowcharts with a particular word length, etc.

Flowchart objectives

Now we have ample information to construct flowcharts. But we face difficult questions:

(1) What are the design objectives?
(2) Which objective is most important? Which one is next? Least?

Here are some reasonable design objectives:

- Limit controller size to some fraction of a single chip. (Since profit goes up as die size goes down, there will be pressure to make the controller smaller even when it fits.)
- Make the machine as *fast as possible* (and certainly faster than its contemporaries).
- Complete the project early—to give the product an early start in the market.
- Make the flowcharts easy to translate into hardware.

This illustrates the value of a good project manager. He ranks the objectives!

Flowcharts

It is time to begin the flowcharts.

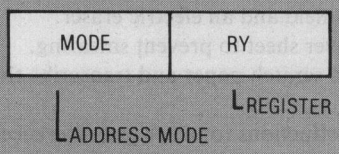
How do we begin?
What do we write?
How do we write it?

I will suggest methods that have worked for me. I use a register transfer notation to describe the operations of the execution unit. Each statement in this notation is called a *task* in the flowcharts. Each *state* contains one or more tasks. Use rectangles for states. (In a microprogrammed

SOME OPERATIONS

ADD
AND
BZ (BRANCH IF ZERO BIT IS SET, REGISTER INDIRECT ONLY)
LOAD (EA IS SOURCE AND RX IS DESTINATION)
POP (POST-INCREMENT WITH REGISTER INDIRECT ONLY)
PUSH (PRE-DECREMENT WITH REGISTER INDIRECT ONLY)
STORE
SUB
TEST

EFFECTIVE ADDRESS



SOME ADDRESS MODES

AB—BASE PLUS DISPLACEMENT. RY PLUS SECOND INSTRUCTION WORD (MODULO WORD SIZE) IS AN OPERAND ADDRESS.

AI— REGISTER INDIRECT. RY HOLDS AN OPERAND ADDRESS.

AR—REGISTER DIRECT. THE RESULT IS STORED IN RY. FOR TWO OPERAND INSTRUCTIONS RY IS ALSO AN OPERAND SOURCE.

Figure 2. MIN instruction set summary.

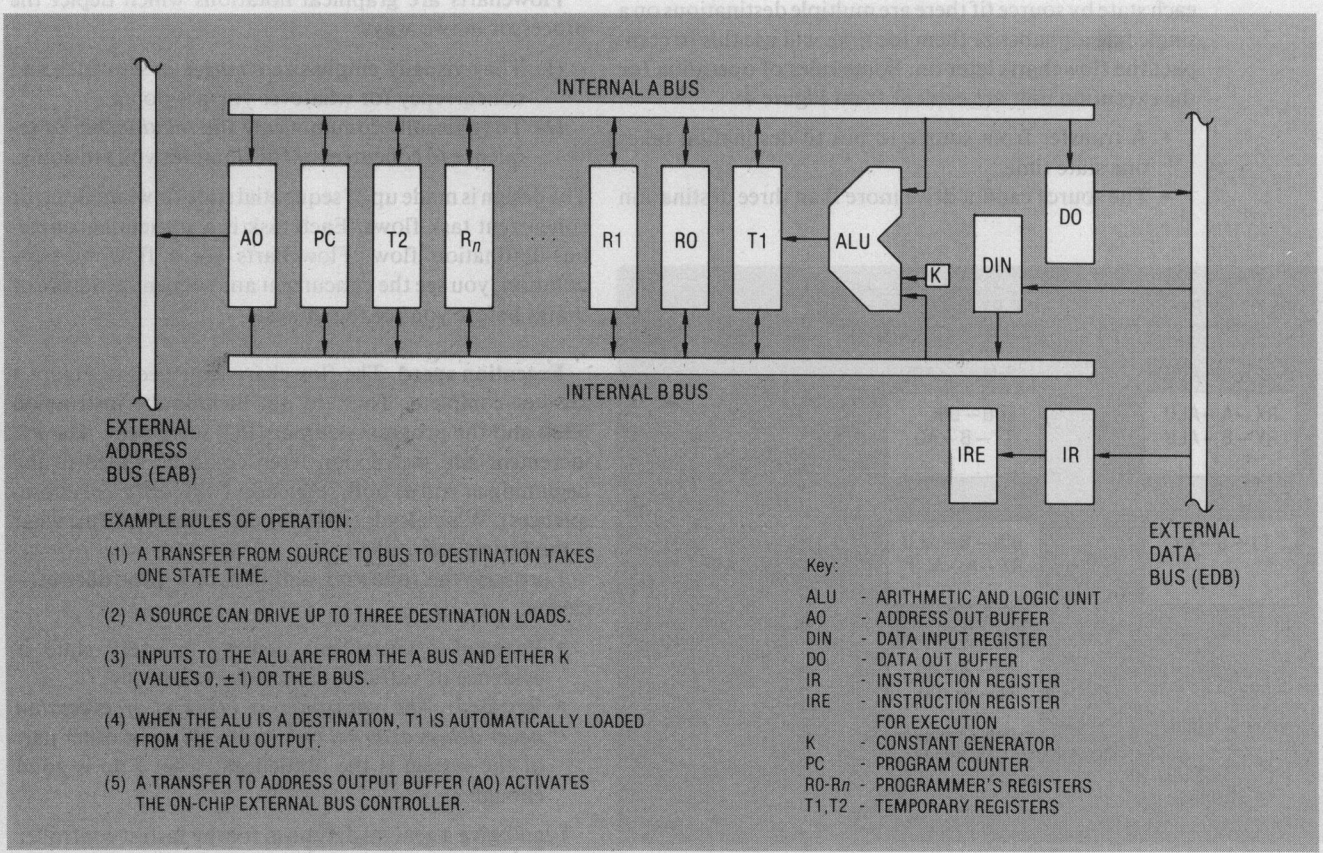


Figure 3. MIN execution unit block diagram.

controller, each state becomes a microword.) A *sequence* is a succession of states. Work on large sheets of high-quality quadrule paper (preferably 17 × 22-inch vellum with 10 lines per inch). Large sheets make it simpler to see and to plan large segments of the control flow, and high-quality paper lasts through many changes.

It takes months to complete the flowcharts for a complicated controller. So you won't have to copy several generations of flowcharts, observe these rules:

- Work in pencil. (Use a 0.5m/m Pentel with F lead.)
- Work on the back of the vellum so you won't erase the grid.
- Use an erasing shield and an electric eraser.
- Always use a cover sheet to prevent smearing.
- Plan changes on scratch paper and transcribe them to the vellum.
- Always use reproductions for work and reference. (I reduced the copies to 8.5 × 11-inch for easier use.)
- Accumulate changes (in red ink) on a reproduction.
- Do trial level 2 flowcharts (level 2 flowcharts are explained later) on 8.5 × 11-inch scratch paper with 1.5-inch high, 2-inch wide penciled-in rectangles as guides.

Figure 4 shows flowchart sequences for the register-to-register ADD instruction and the register-to-memory ADD instruction using the MIN execution unit (Figure 3) and a simple register-transfer notation. Each box is a state. Each line entry in a state is a task. Tasks are expressed in the register-transfer notation; the notation has a source-bus-destination format. Alphabetize tasks in each state by source (if there are multiple destinations on a single line alphabetize them too); we will use this to compact the flowcharts later on. Some rules of operation for the execution unit are evident from Figure 4:

- A transfer from source to bus to destination takes one state time.
- The source cannot drive more than three destination loads. (This information comes from the circuit design engineer.)
- ALU inputs are from the internal A bus *and* either K or the internal B bus.
- When the ALU is a destination, register T1 is automatically loaded from the combinational ALU output at the end of the state time.
- Any transfer to the address out (AO) buffer (see Figure 3) signals a data transfer to the (on-chip) bus controller for the external bus. This bus controller postpones the next state until the external transfer is complete.

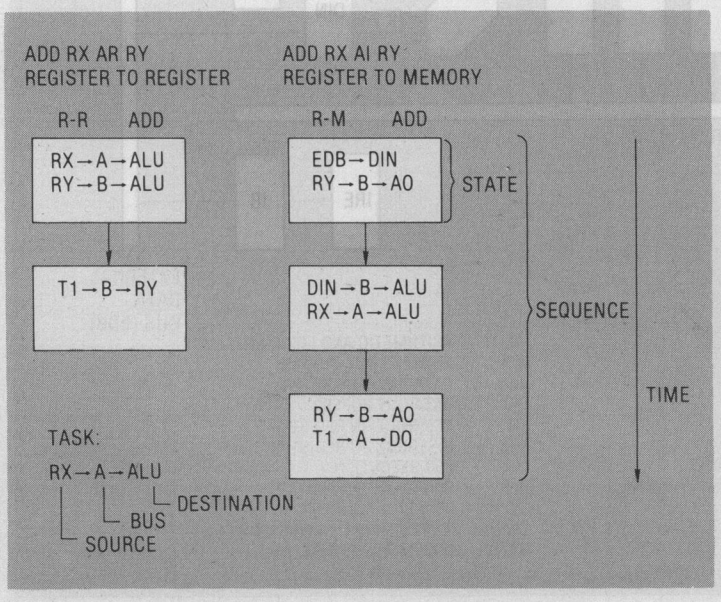


Figure 4. Execution of register-to-register ADD and register-to-memory ADD instructions.

In Figure 4, time advances from the top of the page to the bottom of the page, except within a state. Tasks within a state appear to be concurrent and are governed by rules-of-operation timing. In a microprogrammed controller each state is one microcycle (and may have phases such as source, transfer, destination, and precharge).

The notation of the flowcharts. Keep the register-transfer notation simple! It must capture the essence of what the machine is doing without all the details—which come later. You may think this is a simple notation invented for just this one case. Well, that's somewhat true. The notation is modified to fit the problem. If I want to do a special task, I modify the notation to fit. I want the notation to be a simple, natural, readable way to express what the machine is doing. The notation is not formally defined. In a formal notation, constructs might prevent natural expression of tasks and hinder the design.

Flowcharts are graphical notations which depict the processor in two ways:

(1) They visually emphasize *changes in* sequence and concurrency for whatever you are doing.
(2) They visually communicate the *relationship of* sequence *to* concurrency for whatever you are doing.

The design is made up of sequential state flows made up of concurrent task flows. Each task is a sequential source-bus-destination flow. Flowcharts are a flow-intensive notation; you see the concurrent and sequential nature of things before you see the things.

Execution speed. The flowchart sequences in Figure 4 are not complete. They do not include the instruction fetch and the program counter (PC) increment. The PC increment and instruction fetch could be added to the beginning or end of both sequences (with different consequences). Which leads to the fastest controller? Just what is the fastest controller?

I propose the following definition for controller efficiency:

- Relevant external bus activity in every state is evidence of sufficient controller efficiency.
- Restated: *The controller is efficient if execution never delays external bus cycles.* (If some other part of the system is the bottleneck, what I do is good enough.)

I can't give a general definiton for the fastest controller because I think it depends on what the controller does. I have given a definition that works for a microprocessor;

but this definition is not the one an application engineer would use, because he won't want the external bus tied up by the processor all the time.

Figure 5 augments the examples of Figure 4 with the PC increment and instruction fetch. I removed the lines connecting boxes because they are unnecessary and it saves space. The more states you fit on a page, the more of the design you take in at a glance. (I still use lines to show the next states of sequences with internal branches.) To make a quick measure of efficiency possible, I put a shaded box in the upper right-hand corner of states with external bus activity. Assuming states of equal duration, the overall efficiency of the execution unit is 20 percent for the register-to-register instruction and 50 percent for the register-to-memory instruction. Our competitors will be pleased. What can we do about it? In some states of each flowchart sequence the major internal buses (A and B) are not both occupied. Not good! It should be possible to merge tasks for greater efficiency. We must find a way to squeeze more performance out of the execution unit.

Operation tasks and housekeeping tasks. I separate an instruction's execution into operation tasks and housekeeping tasks and treat each differently. Operation tasks are transfers required to perform the instruction. These tasks (such as accessing operands, storing results, and moving data to and from the ALU) must occur in a specific order and may be unique to a particular instruction. Figure 4 shows the operation tasks for two types of ADD instructions. Housekeeping tasks, such as incrementing the PC and fetching the next instruction, must be performed for every instruction. We have some leeway in when these tasks are accomplished. I separate kinds of tasks so I can optimize the execution of the operation tasks.

Level 1 flowcharts

Figure 6 shows the flowcharts in a format designed to aid merging of operation tasks and housekeeping tasks for maximum execution efficiency. This is the level 1 flowchart format. For each instruction, operation tasks are in the left sequence and housekeeping tasks are in the right sequence. The objective is to merge the housekeeping tasks with the operation tasks. The direction of the merge is *into* the operation tasks. (We are going to make the housekeeping tasks "disappear" into the operation task sequence.) The order of each column must be preserved in the final sequence (called the *execution sequence*) but housekeeping tasks can be merged with operation tasks wherever reasonable, with some restrictions. (We shall see consequences of this merging later.) We would achieve the most efficient execution (for this execution unit) if we merged the housekeeping tasks with the operation tasks without increasing the number of states in the operation task sequence. Usually, it is adequate to have the number of states in the final execution sequence be significantly less than the total states in housekeeping task and operation task sequences.

Increased speed may not be the only objective of the merge. We should also merge the tasks to create as many identical states (*across* instruction types) as possible. We assume that *a controller with fewer unique states is smaller.*

I added one more thing in Figure 6. IRE is the instruction register for execution (see Figure 3). The IRE can be decoded in the state during which it is loaded. It allows a rudimentary prefetch. The IRE holds the current instruction and drives the register selection decoders (for RX and RY). (It cannot be changed until after the last RX or RY

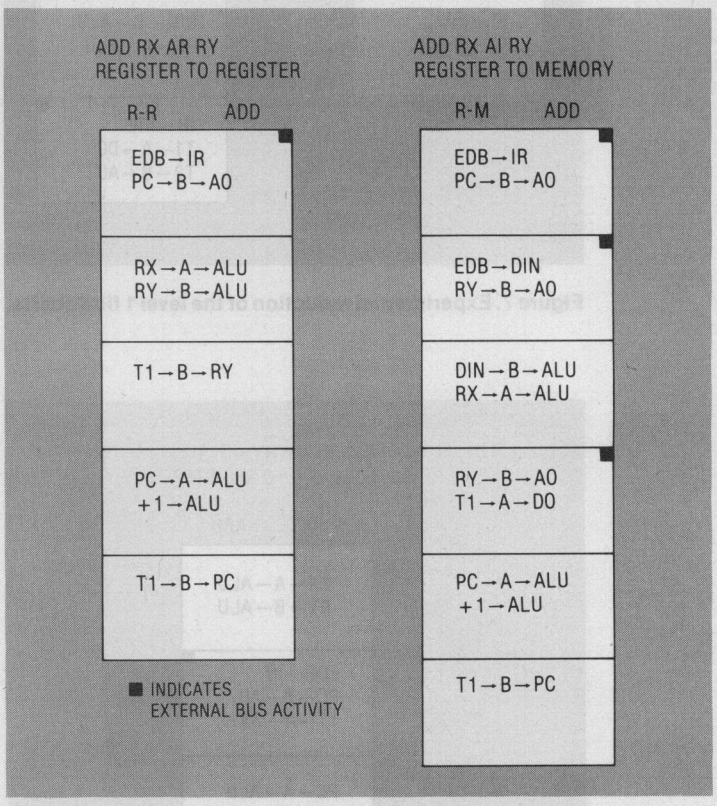


Figure 5. Revised execution of ADD instruction examples.

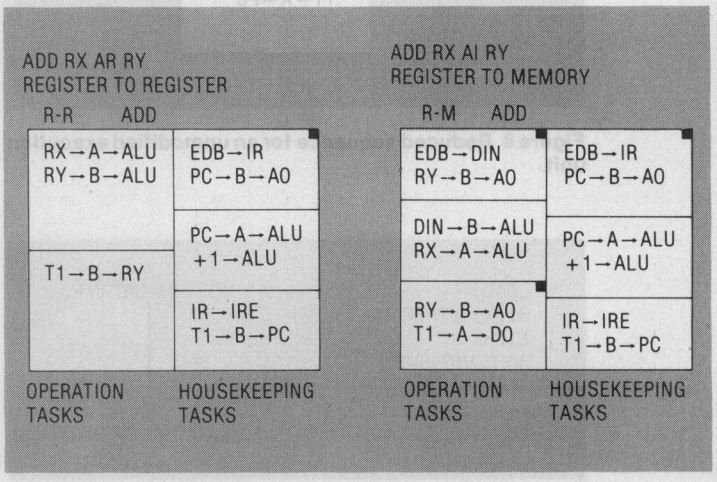


Figure 6. Level 1 flowcharts for two types of ADD instruction.

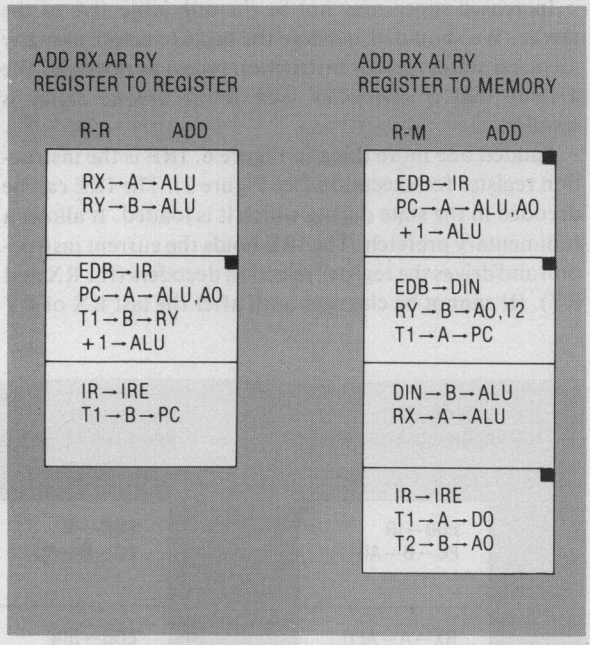


Figure 7. Experimental reduction of the level 1 flowcharts.

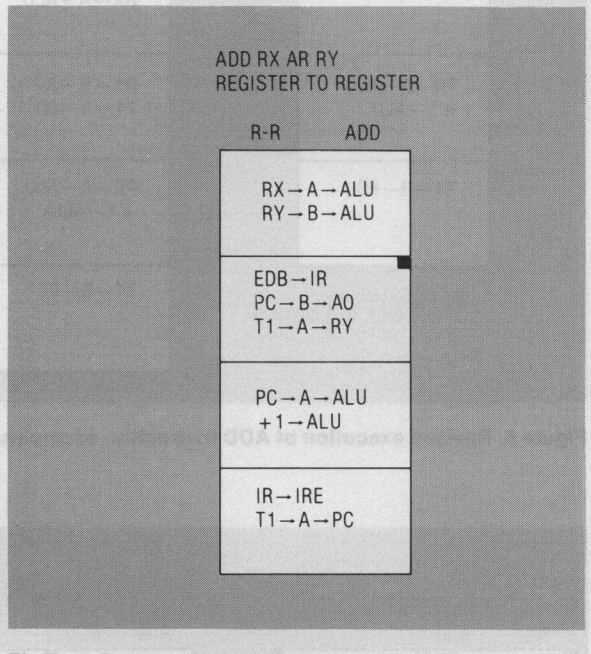


Figure 8. Reduced sequence for an unmodified execution unit.

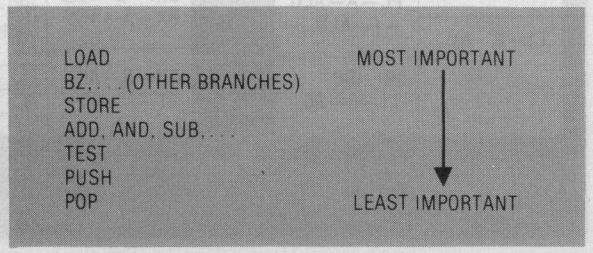


Figure 9. Ranking of MIN instructions.

reference in the flowchart sequence for the current instruction.) The instruction register (IR) can be used to hold the next instruction until the current instruction is done. It can be loaded any time during the current instruction—this is the simple prefetch. More accurately, the IR gets the word following the current instruction (which may not be the next instruction if the current instruction is a branch or a two word instruction).

Level 2 flowcharts

Figure 7 shows the housekeeping tasks merged with the operation tasks to form what I call *level 2 flowcharts*. The efficiency of the register-to-register sequence is 33 percent, and the efficiency of the register-to-memory sequence is 75 percent. (We could do much better by proposing a more complicated machine.) Register T2 saves the operand address in the register-to-memory ADD example. The last state changes the IRE and stores the result because T2 contains the store address and the static decoders (which are driven by the IRE) are available (there are no more RX or RY references).

Feedback on execution unit design. Do a level 2 flowchart of the fastest instruction. This will point to inadequacies in the execution unit design. In general, we discover inefficiencies in the structure of the execution unit as we merge the housekeeping tasks with the operation tasks. In the register-to-register ADD example, if the AO buffer had not been accessible from the A bus (see Figure 3), we would not have been able to do the instruction in under four states. Less than full use of the A and B buses in the resulting sequence would signal the need to improve the execution unit. Figure 8 shows a register-to-register ADD sequence for an execution unit with no path from the A bus to the AO buffer. But beware! The increased complexity of the execution unit can increase the number of unique states and result in a larger controller. So it is only after carefully studying the flowcharts and the execution unit that we suggest execution unit changes to improve the efficiency of the overall design.

Feedback on controller design. Use the format in Figure 6 to create level 1 flowcharts for the entire instruction set. How many sequences is that? The upper bound is 2**w if *w* is the instruction length in bits. However, this is too many because we write only one sequence for each instruction—independent of which registers are specified. (This is an advantage of static decode for the register fields.) We really only need to decode the operation code and the mode bits in the effective address field (see Figure 2). Suppose our simple MIN processor has *k* operations (ADD, AND, OR, SUB, . . .) and *a* address modes (register indirect, base plus displacement, indexed, . . .). If any address mode is valid for any operation we would need *k*a* instruction sequences! Clearly, this number can be large. For example, the Motorola MC68000 has about 14 address modes and over 50 instruction types. If an average instruction has eight states, then we must implement more than 5600 states in the controller. Our chip would make a good office partition.

Idea: If most address modes can be used with most operations, why not share address mode sequences? (Address mode sequences are sequences which only do address calculations.) Then we need only $k+a$ sequences, and that is in keeping with our goal to reduce controller size. It's a good idea, but what will it cost? It's not free. Suppose we enter the execution sequence, jump to an address mode sequence (subroutine), then return to the execution sequence to complete execution of the instruction. Such a subroutine call costs time, but it permits a much smaller controller (since the address mode sequences are shared by most operations). The size and speed goals conflict, so a tradeoff is in the offing.

How important is the time lost in these subroutine calls? To find out, have the instruction set designer rank the instructions in order of importance. He could base the ranking on static or dynamic frequencies of occurrence. However he does it, if he designed the instruction set, he must take the stand on what is important. The ranking for the sample MIN instructions is in Figure 9.

We know sharing sequences reduces controller size. From the ranking we see that slow subroutine calls are costly because at least three of the four most important instruction types can use any address mode. We won't use subroutine calls. Assume address mode sequences can be shared by initially entering the address mode sequence and then branching directly to the appropriate execution sequence. One way to do this in a microprogrammed controller is to have the instruction decoder provide more than one micro address—one for the address mode sequence and one for the execution sequence. Flowcharting has led us to a functional requirement for the controller. (The instruction decoder is to provide more than a single output.) This shows how controller requirements come from the procedure. We have not, however, constrained the implementation of the controller to be combinational or microcoded; that choice lies in the future. We don't even have a block diagram of a controller. And we don't want one yet because we want the procedure to give us the requirements for the controller independently of what we think a controller should look like. The flowchart method finds requirements for a controller that best fit what the machine wants to do (the specification).

Doing level 1 flowcharts

The level 1 flowcharts for the subset of MIN instructions are shown in Figure 10. In a real machine, the flowcharts have many more address mode and execution sequences. Note the following things in Figure 10:

(1) The register-to-register instructions have execution sequences which are not sharable with execution sequences for memory reference instructions. This reduces the savings from sequence sharing.

(2) The flowchart sequences for standard dual-operand instructions (ADD, AND, SUB, etc.) are identical except for the ALU function. They can use the same execution sequence if you use the op code directly to specify the ALU operation (the same way register fields drive the register selection).

(3) Unfortunately, the STORE instruction reads the word at the store destination location because it shares the address mode sequences with other instructions. We sacrificed speed to make the controller smaller. (There are other reasons you may not want to do STORE with a read first. Some systems want locations that are read protected. Other systems have memory-mapped I/O peripherals that change states upon read.)

(4) The branch-on-zero (BZ) instruction is a special case. Since the condition code (Z) may have been set on the last state of the previous instruction (in TEST, for example), it may not be available to help make the next state decision. (Because of the simple prefetch, the instruction decoder can be operating concurrently with the execution unit. Information that can change in the execution unit cannot be used by the instruction decoder.) As a result, the branch appears between the first and second states of the flowchart sequence for the branch instruction. Because there is a delay state for decision, we must decide what to do with it. The example in Figure 10 shows an anticipated branch prefetch (it is discarded if the branch is not taken). The instruction set designer should be able to tell you which condition ought to be made faster (or if both must be equal).

Each instruction's flowchart further shapes the design. Functional characteristics of the controller will eventually be completely defined by the flowcharts. We have still not constrained the design to be either combinational or microprogrammed. Once we combine the standard dual operand instructions (ADD, AND, SUB, . . .) into one flowchart sequence for register-to-register and another for register-to-memory, the level 1 flowcharts are complete. Then we can work with the level 1 flowcharts, merging housekeeping tasks with operation tasks to produce the level 2 flowcharts.

Doing level 2 flowcharts

Figure 11 shows the housekeeping tasks merged with the operation tasks for the instructions in Figure 10. I try to do this without increasing the number of states in the operation task sequence. In Figure 10 I wasn't able to do this; I did reduce the number of states from a potential 54 to an actual 35. (I merged 28 housekeeping states with 26 operation states.) If I can't reduce the number of states significantly (a matter of judgment), I try to improve the execution unit. Take care in merging, because operation tasks can use the same resources (such as buses, registers, and arithmetic units) as the housekeeping tasks so arbitrary interleaving is not possible. For example, if there are PC (program counter) relative address modes, the PC update (a houskeeping task) must consistently precede (or follow) the address calculation in all of them. If a problem during an instruction execution causes an interrupt that stores the old PC value, that value should be consistent for all instructions.

Does this complete the flowcharts? Not quite. Before we can transform the flowchart description to a hardware implementation, we must identify the states. In Figure 11 I put state identifiers in the lower right-hand corner of each state.

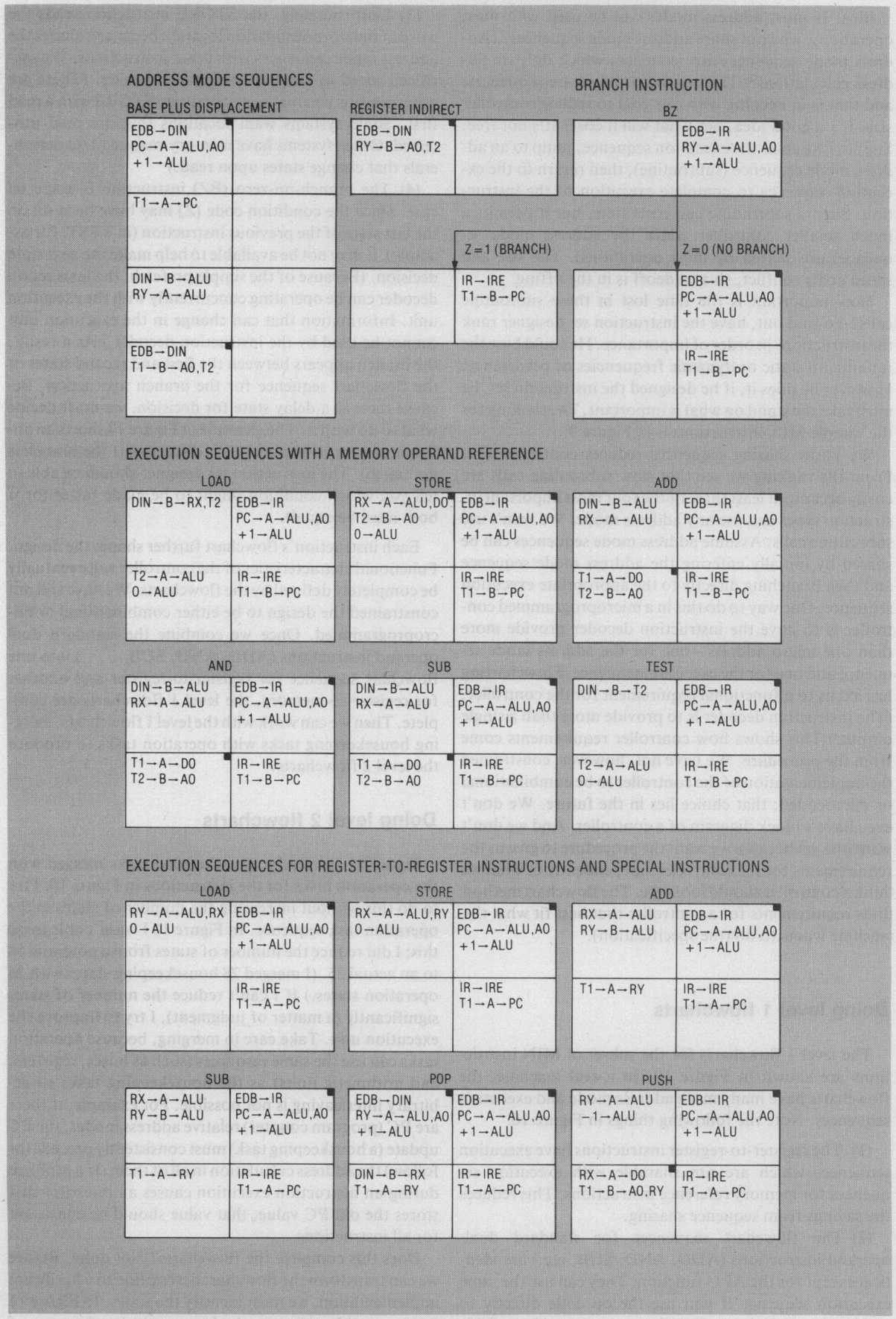


Figure 10. Example level 1 flowcharts for the MIN processor.

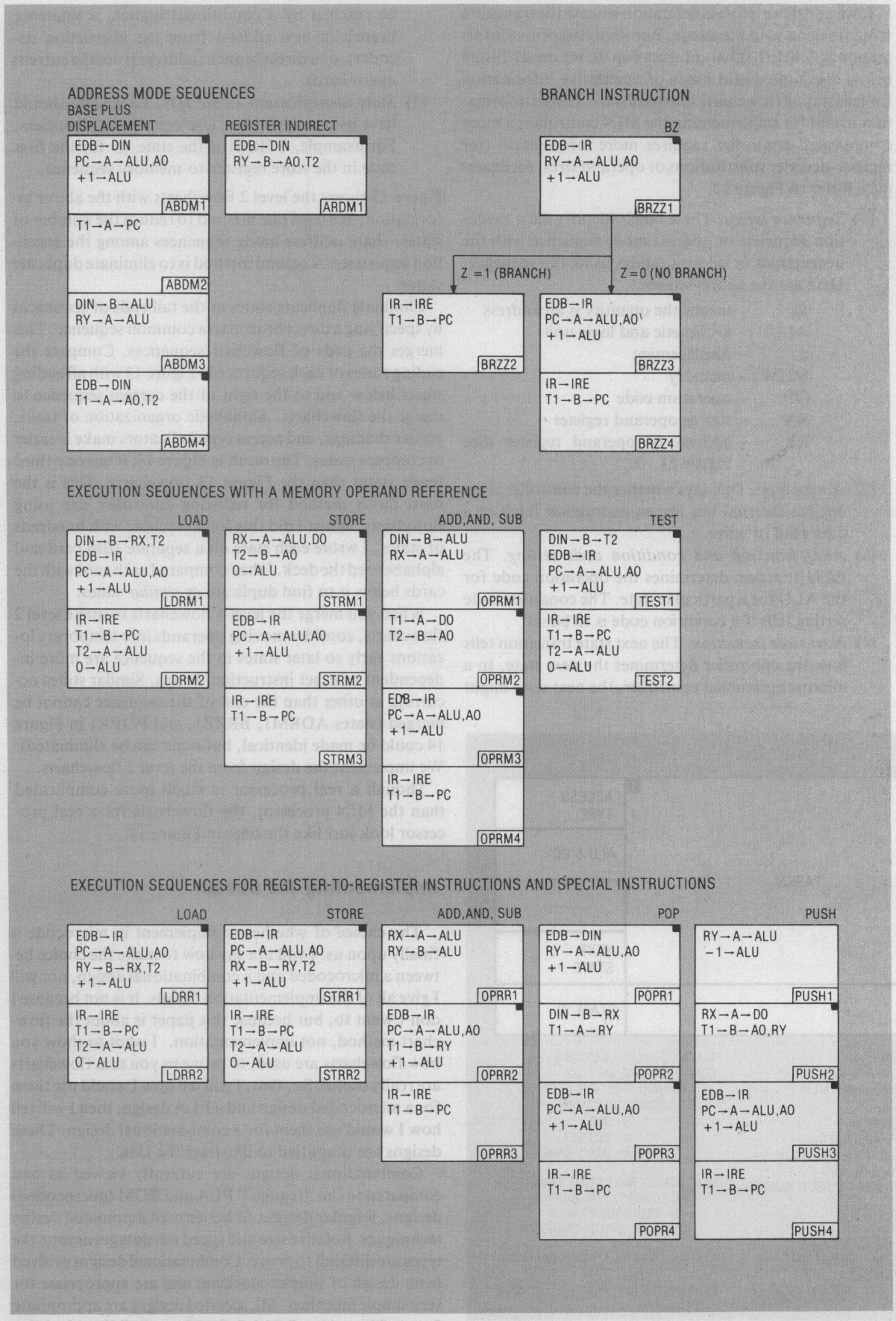


Figure 11. Merged level 1 flowcharts for the MIN processor.

If we add descriptive information we ease the transition from flowcharts to hardware. But what descriptive information will help? What information do we need? Listed below are some useful kinds of descriptive information for translating flowcharts into hardware. I listed information useful for implementing the MIN controller; a more complicated controller requires more information (for register-decoder substitutions or operand sizes, for example). Refer to Figure 12.

(1) *Sequence labels.* These labels identify each execution sequence or address mode sequence with the instructions or address modes using the sequence. Here are the abbreviations:

@	- means the quantity is an address
ALU	- arithmetic and logic unit
d	- displacement
MEM	- memory
OP	- operation code
RX	- source operand register
RY	- address or operand register (See Figure 2)

(2) *Access type.* This says whether the controller is using the external bus for an instruction fetch or a data read or write.

(3) *ALU function and condition code setting.* The ALU function determines the operation code for the ALU for a particular state. The condition code setting tells if a condition code is to be set.

(4) *Next state transition.* The next state transition tells how the controller determines the next state. In a microprogrammed controller, the next state might be reached by a conditional branch, a sequence branch (a new address from the instruction decoder), or a direct branch (address from the current microword).

(5) *State identification* (*state ID*). Each state should have its own identifier. Use descriptive identifiers. For example, STRM1 is the state ID for the first state in the store register-to-memory sequence.

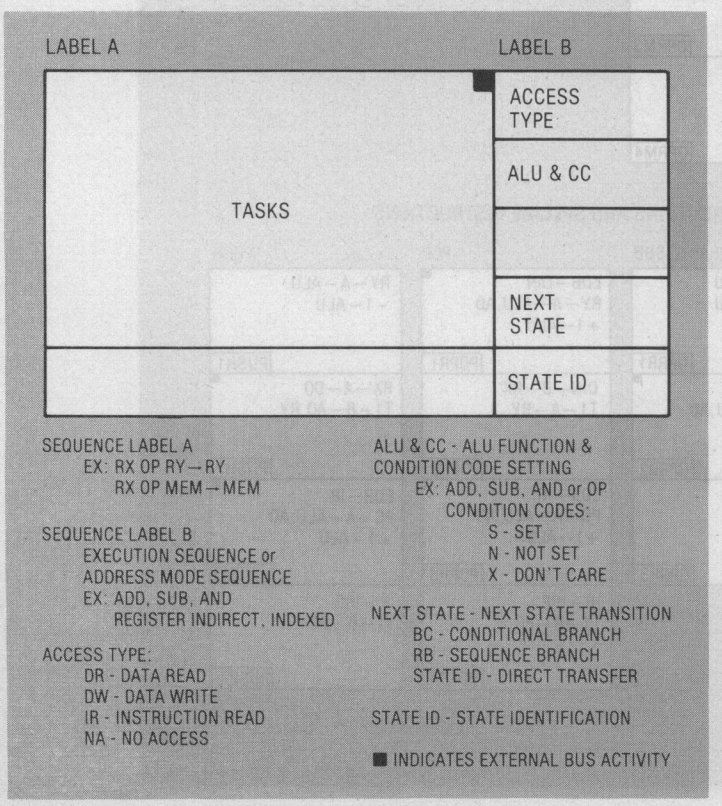


Figure 12. Template for a level 2 flowchart state.

Figure 13 shows the level 2 flowcharts with the above information. We used one method to reduce the number of states: share address mode sequences among the execution sequences. A second method is to eliminate duplicate states.

Eliminate duplicate states at the tail ends of sequences by specifying a direct branch to a common sequence. This merges the ends of flowchart sequences. Compare the ending states of each sequence in Figure 13 with all ending states below and to the right of the current sequence to merge the flowcharts. Alphabetic organization of tasks, corner shadings, and access type indicators make it easier to compare states. The result is Figure 14; it has one-third fewer states than the Figure 13 flowcharts. This is the most direct method for reducing controller size using flowcharts. When I did this for a machine with hundreds of states, I wrote each state on a separate IBM card and alphabetized the deck. I then compared each card with the cards below it to find duplicate or *similar* states.

When you merge the level 1 flowcharts to make level 2 flowcharts, consider moving operands into temporary locations early so later states in the sequence are more independent of exact instruction details. Similar states occurring at other than the end of the sequence cannot be merged (states ADRM1, BRZZ1, and POPR1 in Figure 14 could be made identical, but none can be eliminated). We implement the design from the level 2 flowcharts.

Though a real processor is much more complicated than the MIN processor, the flowcharts for a real processor look just like the ones in Figure 14.

Implementing the design

The choice of whether to implement in microcode is finally upon us. I will not say how to make the choice between a microcoded and a combinational design, nor will I give all of the implementation details. It is not because I don't want to, but because this paper is about the flowchart method, not implementation. I want to show you how flowcharts are used, to prove to you that flowcharts are really useful. So, first, I will tell how I would use them for a microcoded design and a PLA design; then I will tell how I would use them for a combinational design. These designs are simplified to illustrate the idea.

Combinational designs are currently viewed as bad compared to the "regular" PLA and ROM (microcoded) designs. Regular designs fit better with automated design techniques. Relative size and speed advantages among the types are difficult to prove. Combinational designs evolved from design of simpler machines and are appropriate for very simple machines. Microcoded designs are appropriate for machines using ROM or RAM as a building block for the controller or for machines that need microprogram-

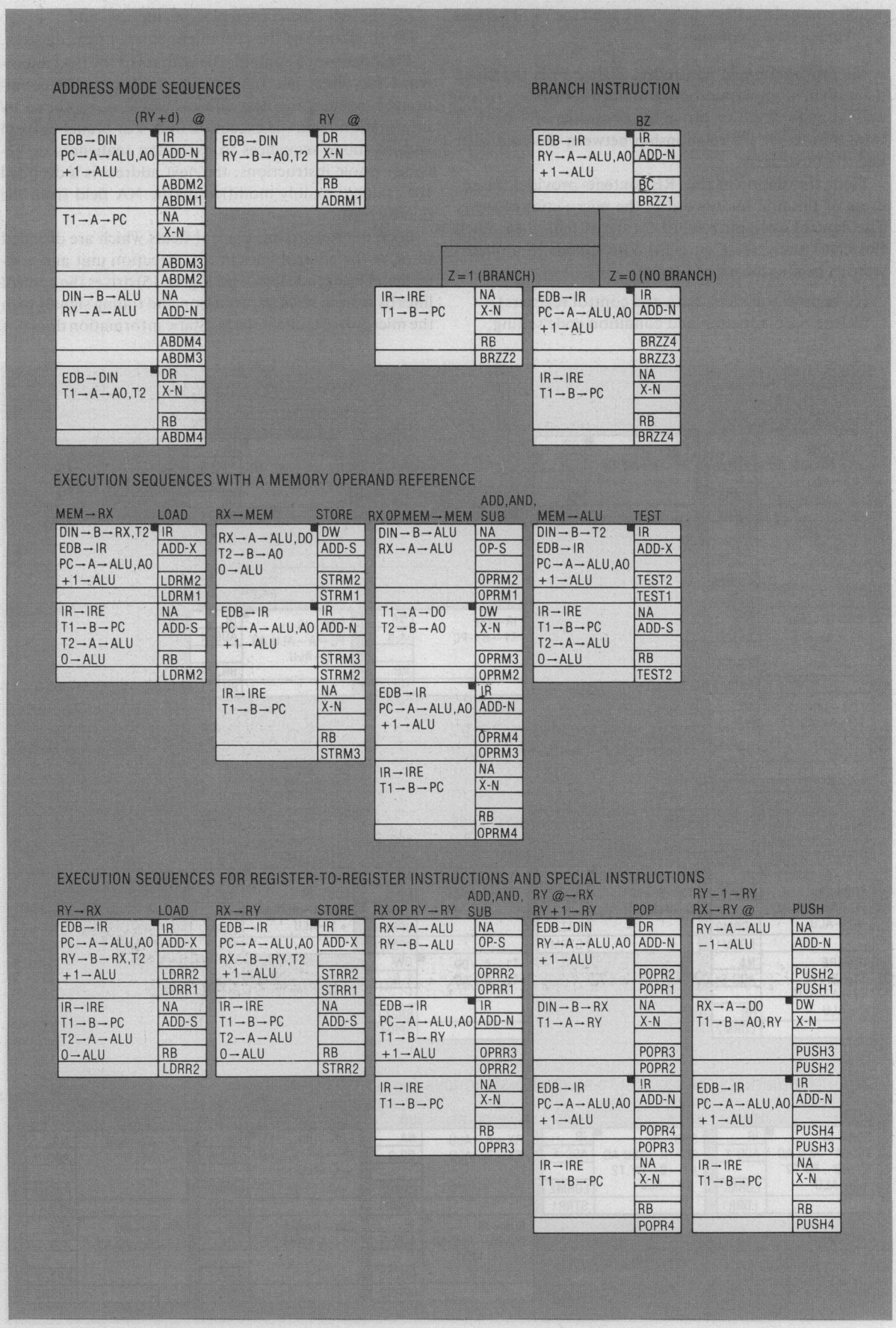


Figure 13. Format for final version of the level 2 flowcharts.

mable controllers. PLA designs are good for VLSI because chip area is at a premium.

Microprogrammed controller. Figure 15 is the block diagram of a simple microprogrammed controller. I show only enough detail to prove the controller will work. I want you to see the relationship between the controller and the flowcharts.

Some translation of the IRE contents provides the address of the first microword in the micro control store. The control store microword format is Figure 16. Each flowchart state (see Figure 14) corresponds to a microword. The microword can specify

(1) register transfers (data and control registers),
(2) the ALU function and condition code setting,
(3) the next micro control store address, and
(4) the source of the next micro control store address.

The microword contains the address of the next microword for direct microinstruction branches; for conditional branches, the next address would be modified by information from the execution unit. For branches from address mode sequences to execution sequences or between whole instructions, the next address is a decoded IRE value (possibly modified by the NA field from the microword).

Each microword has control fields which are decoded to drive the control lines in the execution unit and controller. The decode logic (see Figure 15) drives the control lines by mixing static information and timing signals with the microword control fields. Static information does not

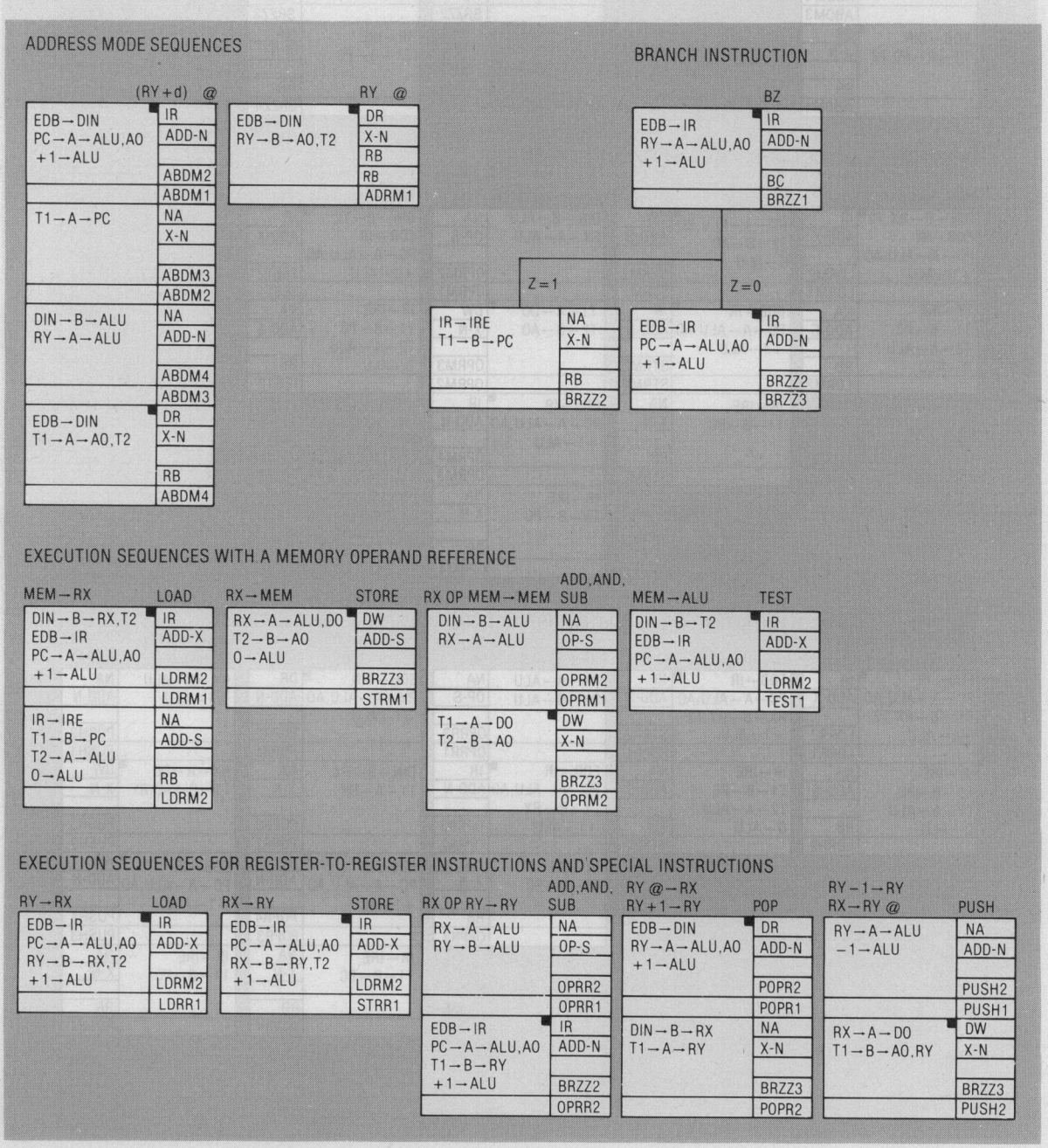


Figure 14. Merged level 2 flowchart example.

change during the instruction execution and can go directly to the decode logic. The register fields in a register-to-register ADD instruction, for example, do not change during instruction execution.

In a simple microprogrammed controller implementation, each state in the level 2 flowcharts corresponds to one microword. Thus, each state in the level 2 flowcharts maps to one word in the micro control store. The fewer microwords we have, the smaller the micro control store will be.

To program the control store we must transform the flowcharts into control store bit patterns.

- The state ID becomes the location of the microword in the micro control store.
- The next state becomes the micro address select (TY) and next address (NA).
- The tasks become bits in the control bit fields.

These transformations can be done on a computer, but you will probably want to assign control store locations to make the address decoder smaller (also, certain control store addresses are reserved for reset, interrupt, and other special sequences).

PLA controller. Figure 17 is a block diagram of a simple programmed logic array controller. Note the strong similarity between the PLA controller in Figure 17 and the microprogrammed controller in Figure 15. I consider PLA controllers to be a variety of microprogrammed controller. If the control store address logic and the micro control store are implemented as an AND-OR PLA, the address logic would be the AND array and the micro control store would be the OR array.

Another way to see the similarity between a microcoded implementation and the PLA implementation is to consider the micro control store as an *orderly decode* of an input address into a microword. If the control store address logic (address decode, micro address modifier, and multiplexer) of Figure 15 produced the microword direct-

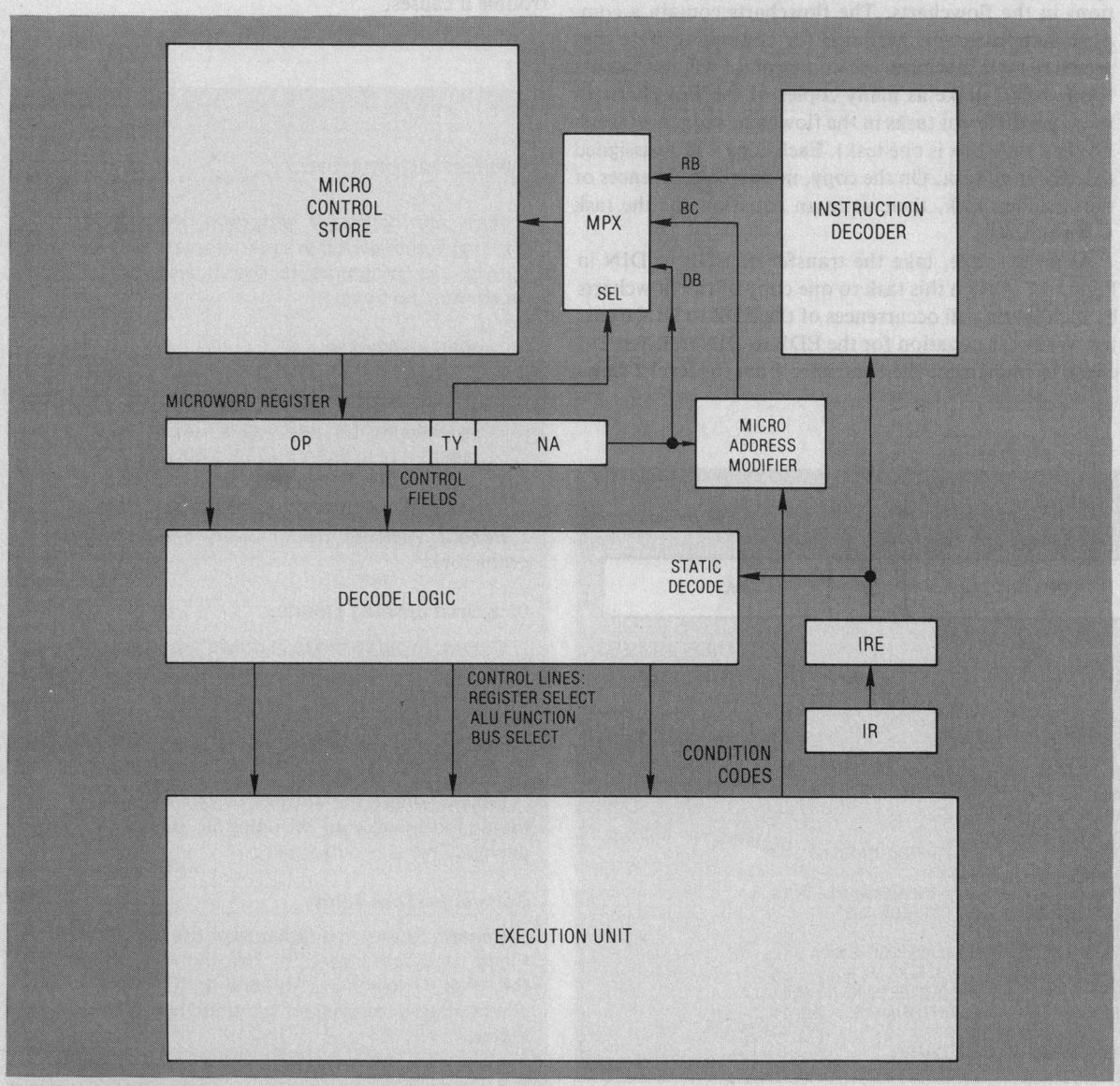


Figure 15. Microprogrammed controller block diagram.

ly (instead of an address for the micro control store); it would behave exactly like a PLA.

The flowcharts are used in the same way except now you may be able to combine like states in the flowcharts not at the ends of sequences (provided the states can be made to lie logically next to each other for the AND decoder). Program the PLA OR array using the same flowchart transformations as for the microprogrammed controller. The PLA OR array contains the same bit patterns as the control store for the microprogrammed controller. Unused control store locations will be left out of the corresponding PLA. A further reduction in controller size may be possible using methods for splitting or folding a PLA. I will not discuss these; they don't relate directly to the use of the flowcharts.

Combinational controller. There are many ways to design a combinational controller (also called a combinatorial or random logic controller). I will describe one. First, design a state machine to duplicate the state transitions in the flowcharts. The flowcharts contain a complete state diagram. Methods for converting state diagrams to state machines are known and I will not discuss them. Next, make as many copies of the flowcharts as there are different tasks in the flowchart sequences (each line in a state box is one task). Each copy will be assigned to a different task. On the copy, mark all occurrences of the assigned task, then write an equation for the task using state IDs.

As an example, take the transfer of EDB to DIN in Figure 14. Assign this task to one copy of the flowcharts by highlighting all occurrences of the EDB to DIN transfer. Write the equation for the EDB to DIN transfer. If I chose to implement the controller from the level 2 flowcharts of Figure 14, the equation for the EDB to DIN transfer would be

$$EDIN = ABDM1 + ABDM4 + ADRM1 + POPR1$$

EDIN is the name given to the control point (gate) controlling the transfer of EDB to DIN. The real equation for the transfer would be much larger if all the flowchart sequences were available, but the method is the same.

If I write the equations for unique states instead of individual tasks, I end up with the OR array from the PLA controller. (Lines duplicated in the OR array will not be duplicated here, but the state controller might be bigger.) This is because PLA design fixes the decode method for all terms (a two-level NAND NOR or a three-level AND OR AND, for example) but combinational design allows the implementation of terms to vary individually. I could write equations for groups of lines or group subexpressions of the equations to reduce the logic. The flexibility of this method is probably the source of some of the trouble it causes.

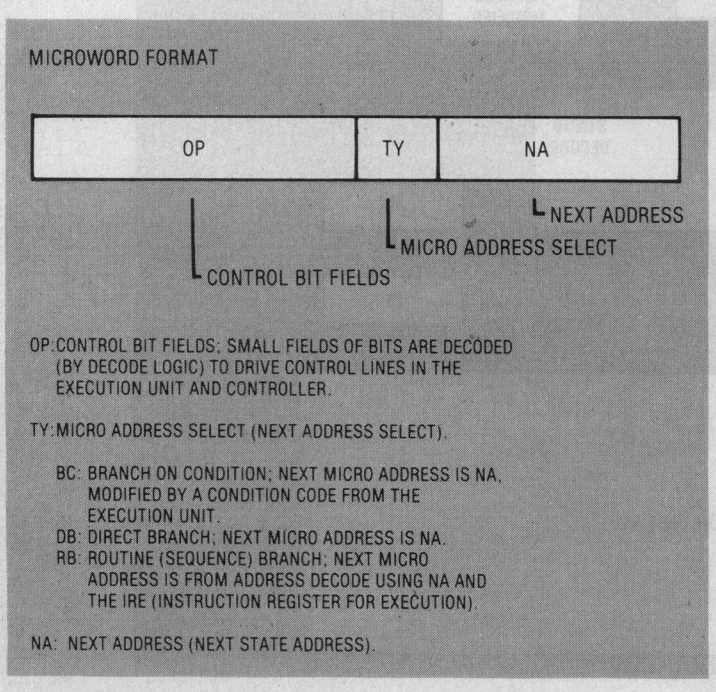


Figure 16. Control store microword format.

Design automation

Here are computer programs that make flowcharting easier (listed in order of usefulness). The input to the programs is the flowchart text (just characters, no boxes).

Flowchart assembler

Output:

- Microcoded or PLA—produces the boolean equivalents for the tasks. (In a microcoded design, these are microword bit patterns.)
- Combinational: produces lists of states for each task. This helps with minimization.

Benefit: Relieves you of a *very* tedious and error-prone task.

Flowchart drawing program

Output: In batch mode, it draws the flowcharts on a high-resolution plotter. In interactive mode, you enter the name of the instruction from a display terminal and the program displays the first state in the flowchart sequence on the screen. You step through the states by just hitting the enter key.

Benefit: Draws the flowcharts and shows state contents (and microword bit patterns, for a microcoded design). This aids debugging.

General purpose editor

Benefit: Scans text faster than the human eyeball. Finds candidate states for merging when converting the level 1 flowcharts to level 2 flowcharts. Finds similar states to consider for reducing the number of states.

Why it works

The flowchart method lets you use your natural human skill at pattern recognition by showing you whole instruction flows and groups of instruction flows together on a single page. With this method, you literally see the structure of the design evolve from a rough proposal to a complete design. You never have a limited set of tasks or sequence of states whose order is fixed as the result of imposing some form of preconceived block diagram of the controller on the design. Rather, you start with a block diagram of an execution unit and simply begin flowcharting the sequence of states for each machine operation. By using the method discussed, you derive the detailed level 2 flowcharts for the controller.

Because these flowchart sequences are unhampered by independent notions of how control signals and data "should" flow, the controller design can emerge in a way precisely suited to the operation of this machine. Whether it *does* emerge depends on how well you make the technical judgments you are faced with every step of the way.

"You mean success with the flowchart method can depend on a bunch of mysterious 'technical judgments'?" you ask.

You bet . . . in the sense that the more proficient you are as an engineer the greater insight you have into the organized picture the flowchart method presents. (To fully automate or even to express fully that part of engineering design is the domain of an artificial intelligence project.)

I've asked many chip designers, "How did you get from the English to the logic design?" To some, the oddness of my query was that I seemed to be asking how *thinking* works. But I wasn't asking that. I was asking what the designers *did*. Among chip designers it takes several minutes to explain what I'm talking about because this level of chip design is rarely discussed. (It actually takes less time among non chip designers because they don't know that.) Until recently, we engineers haven't had to attempt anything complicated! That sounds ludi-

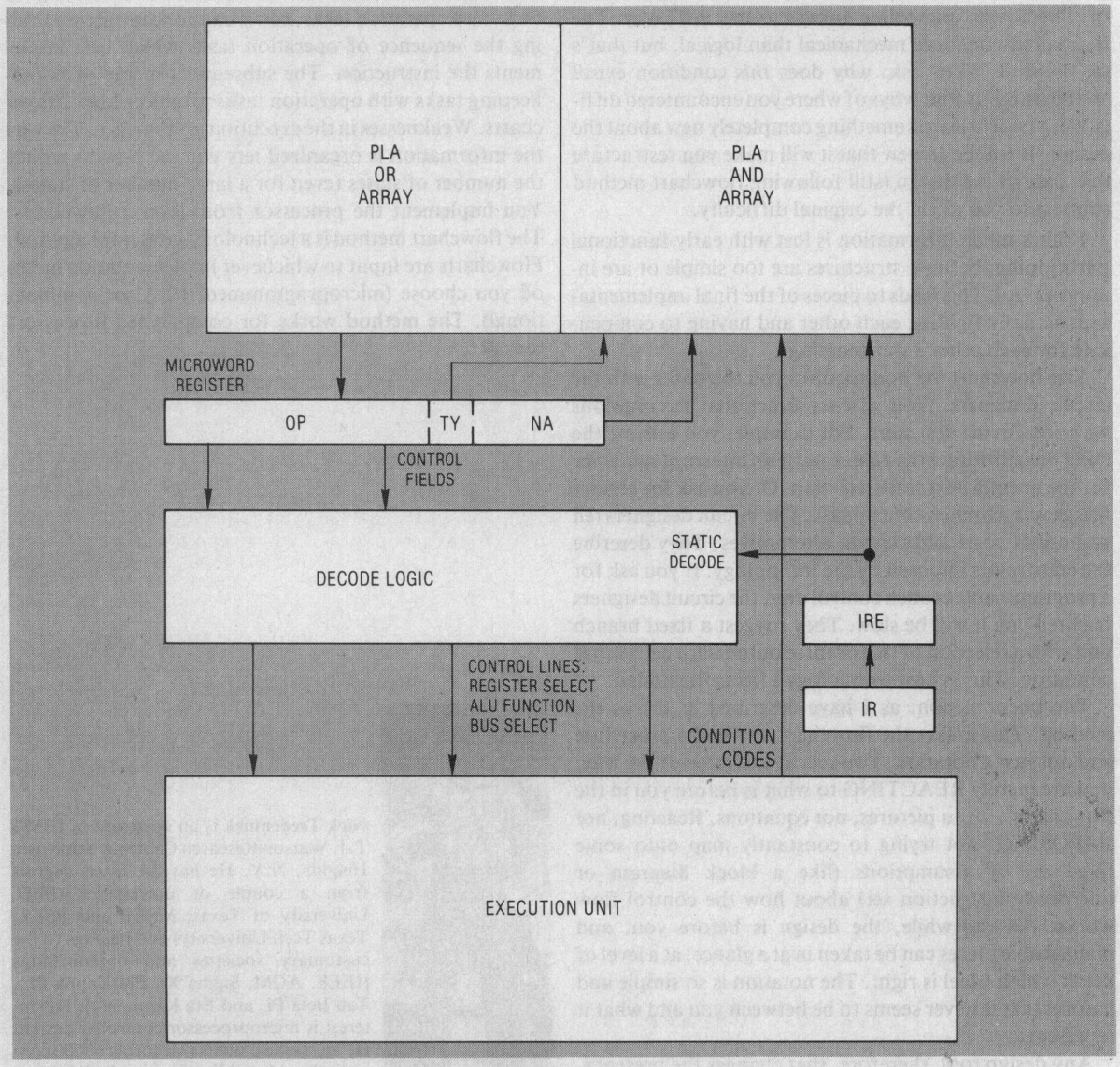


Figure 17. PLA controller block diagram.

crous but it is really true at the level of the initial specification. We design complicated circuits, but what the chip does as a whole is describable in a sentence or two. We don't even come close to the titanic complexities of a large operating system or data base management system—complexities which programmers routinely face.

I've asked many chip designers, "How did you get from the English to the logic design?" To some, I seemed to be asking how *thinking* works.

The flowchart method is not a way to *think*. It is a way to write down the design so that you can see it in an organized way. The value of the method is that in organizing the steps you encounter new insights. You see a new pattern in the logic or, you notice you are having difficulty. There is an immediate answer to this difficulty. The reason may be more mechanical than logical, but that's ok. Note it. Then ask: why does *this* condition exist? Within three or four whys of where you encountered difficulties you will learn something completely new about the design. It will be so new that it will make you restructure that part of the design (still following flowchart method rubrics) so you avoid the original difficulty.

I think much information is lost with early functional partitioning, because structures are too simple or are inappropriate. This leads to pieces of the final implementation actually fighting each other and having to compensate for each other's shortcomings.

The flowchart method requires you to confer with the circuit designers. You discuss functional assumptions with the circuit designers. For example, you assume the ability to substitute the first state of an interrupt sequence for the normal next state selection. Or you ask for certain features in a branch control unit. The circuit designers tell you about costs and suggest alternatives. They describe the constraints imposed by the technology. If you ask for a programmable branch control unit, the circuit designers may tell you it will be slow. They suggest a fixed branch unit with a selection of two or three outputs for each input condition. This is how technology affects the design.

The phenomenon, as I have described it, drives the method. This makes the flowchart method a procedure and not just a notation. You see, at each step of the way, you are merely REACTING to what is before you in the flowcharts . . . in pictures, not equations. Reacting, not IMPOSING, not trying to constantly map onto some fixed set of assumptions (like a block diagram or microcode instruction set) about how the control flow works. All the while, the design is before you, and manageable pieces can be taken in at a glance, at a level of detail which I feel is right. The notation is so simple and natural that it never seems to be between you and what it represents.

Any design tool, therefore, that changes the presentation format or limits what you can see of the flowcharts to just a portion of a page interferes directly with the method and isn't as good as pencil and paper.

Flowcharting is an iterative procedure and it uses feedback from a register-transfer simulator (or breadboard) and a circuit simulator. (A register-transfer simulator accepts the MIN's instructions, assumes the MIN's execution unit, substitutes flowchart sequences for MIN instructions, and pretends to do the tasks in each state.) You compare what your flowchart sequences do with what the user's manual says the instruction should do. The result: you correct errors and add successive detail to the flowcharts. If you didn't correctly update the stack pointer to point to the next empty location, a register-transfer simulator will quickly point to the problem. Even if an instruction appears to work, a register-transfer simulator finds subtle bugs underlying context-dependent errors.

The flowchart method is a straightforward way to derive the design of a processor. The level 1 flowcharts distinguish two types of activities, called housekeeping tasks and operation tasks, so we can concentrate on finding the sequence of operation tasks which best implements the instruction. The subsequent merge of housekeeping tasks with operation tasks produces level 2 flowcharts. Weaknesses in the execution unit surface. The way the information is organized lets you see how to reduce the number of states (even for a large number of states). You implement the processor from level 2 flowcharts. The flowchart method is a technology-conscious method. Flowcharts are input to whichever implementation method you choose (microprogrammed, PLA, or combinational). The method works for complicated processors too. ■

Nick Tredennick is an occupant of IBM's T.J. Watson Research Center in Yorktown Heights, N.Y. He has the usual degrees from a couple of universities (PhD, University of Texas; MSEE and BSEE, Texas Tech University) and belongs to the customary societies and organizations (IEEE, ACM, Sigma Xi, Phi Kappa Phi, Tau Beta Pi, and Eta Kappa Nu). His interest is microprocessor controller design. He has some patents (Motorola 68000 controller) and publications and claims an average number of jobs for experience.