Contents

CHAPTER

1

Here's the Deal

This book is for graduate level electrical engineering or computer engineering students. "Graduate level" means I assume you are proficient at, not just knowledgeable of, basic logic design. You can reduce a five-variable Karnaugh map to minimized logic in a few minutes. You can "read" multilevel NAND-NAND logic diagrams for output functions. And you can look at the logic diagram of a simple machine and see how it works. I would like you to read this book and somehow go beyond a procedure. Steal ideas. I describe the Micro/370 microprocessor in detail, but I am really talking about design ideas, using Micro/370 as an example. I want you to add these ideas to your design repertoire. Make them the stepping-off point to your own design experiences.

I present an industrial logic design method for single-chip microprocessors, called the flowchart method. I do this using a real example. The case study is Micro/370—a single-chip System/370 microprocessor designed using the flowchart method. Micro/370 consists of about two hundred thousand transistors (sites). I wrote this text as I did the logic for Micro/370. I also used the flowchart method when designing the logic for the Motorola MC68000.

Books describe methods as if they are step by step in practice. But methods are not step by step. There are always problems. Students lose confidence when they are unable to apply a method as cleanly as it is described. I present the flowchart method both ways—the tutorial and the dirty reality. I discuss mistakes in this book because mistakes are an integral part of what you do.

I intentionally repeat things from chapter to chapter; I do this for emphasis and to gradually introduce detail.

Why I Think This Description Is Important

This description tells how an engineer actually works. If you are a new logic designer, think of this as a way to get started, an organized way to develop your own style. This is a documented industrial logic design method, which means I wrote what I think you should do, in detail, to design the logic of a microprocessor.

The problem with many texts is that we lie about details. We are sloppy in areas that are not our primary concern. Academics are method fanatics. Practitioners are solution fanatics. In school, we glorify the methods and lie about the sophisticated problems we solved. In industry, we glorify the problems and lie about the sophisticated methods we used. Each side loses credibility the minute one side reads the other's literature. The academic knows that the practitioner's "method" is (ugh!) arbitrary, just as the practitioner knows the academic's "solution" is (ugh!) not applicable. Because we oversell method and solution, it takes too long to figure out what really works. This book puts what I think really works for microprocessor logic design in one place.

Using Computers

Eventually, you will enter your design into computer files. Lots of people have tried to make this part "easier." These people are called design automation (DA) experts. Designers work on designs, and DA people work on automating design. As a designer, my view is that DA should *support* design, not *be* design. After I have designed something, I think, "Boy, it would be nice if this part (of the way I design) were automated in this particular way." But sometimes I think DA people automate things and

want designers to design in terms of inputs and outputs to their design tools. You may feel frustrated when you ask a DA person, "Why doesn't your program let me do this?" and the response is, "Why are you doing it *that* way?"

Imagine this: You start a new job at a company, and on day 2 they say, "Here's where you enter your logic—in our Humongous Design System (HDS)."

Surprised, you say, "But I've only had eight courses in digital design principles. How do you actually *design* a microprocessor? I mean . . . come up with the logic for something that complicated . . . in an organized way?"

"Well, you partition the problem into manageable pieces," they reply.

Stubbornly you ask, "But how do you know what the right pieces are?"

Perspiration is forming on your brow. They've found you out: You were supposed to have learned this in one of those courses. But your host merely replies, "Oh, that's easy, you just piece the logic together from the structures that are good for this technology. They're right here. See, here's a sixteen-way NOR, and here's a three-input NAND and . . ."

You don't hear much of the speech. You have no choice: You must use HDS because it automatically verifies your logic; selects, places, and wires the circuits; and generates test patterns using a fault model that has been accepted companywide. Besides, output from HDS is the only output manufacturing will accept. Period.

In a case like this, designers start thinking of solving problems in terms of how to express the solution in a particular notation. They structure the solution out of only the conceptual constructs supported by that notation. (If the tool does not support pass gates, guess what—no pass gates in the design.) In this way, DA becomes design.

This book is partly in response to the growing presumption that computers are an essential part of logic design. They are not. I want to describe here the essence of a logic design method.

Computers are not an essential part of this flowchart method. I think of computers as an expensive and awkward alternative to pencil and paper. Even so, I don't think *all* DA is bad. When DA's good, it's very, very good, but when it's bad, it's horrid.

The Right Structures

The key to performance in microprocessor design is finding the right conceptual structures. How do you best represent the concepts present in the architecture document? If you are constrained by notation, by the contents of a circuit library, or by the fact that wiring must be on a grid, then you lose a lot of performance. (I make this flat statement with no proof.) I believe that basing your design on the right conceptual structures—ones that make the design "flow"—is the key to high performance. I do not believe it matters (in performance) whether these structures are implemented in programmable logic arrays (PLAs), read-only memory (ROM), or random logic. ROM- or PLA-based logic is certainly easier to change physically, but neither is inherently faster or slower than random logic.

You will frequently encounter statements such as, "Microcode leads to slower control paths and adds to interpretive overhead," and "Hardwired control provides for the fastest possible operation." These statements are not true. What *is* true is that microcoded solutions tend to be used for interpretive structures, and interpretive structures are slower. For those who believe "microcoded" means "interpretive," think of what I am discussing as microcoded implementations that are not interpretive (a seeming contradiction).

Speed comes from using the right logical structures for the job. Structured garbage logic is still garbage logic. How do you find the right logical structures? Use the flowchart method. It gives you a framework (notation and procedure) that organizes design details so that you can see logical patterns. After that, everything depends on how good you are at logic design.

Chapter 8 shows you where the flowchart method fits in relation to other methods of logic design. I prefer microcoded designs. (See Appendix A for my definition of microcoded. To me, the word "microcode" is interchangeable with "microprogram.") The examples I use, real and contrived, are mainly microcoded ones.

C H A P T E R

2

Defining a Microprocessor

A microprocessor is a computer's central processing unit (CPU) implemented on a few (say, fewer than four) silicon chips. The processor has two parts: a control part and a data part. The control part says what to do, and the data part does it. The control part decodes instructions and guides the processor through its internal states. The data part (or execution unit) contains the registers, arithmetic units, shifter, and other pieces that directly store or manipulate data. The control part directs operations in the execution unit. It consists of the clock-phase generators, bus controller, and processor controller. The processor controller consists of a control store (with all the microcode), state sequencer, instruction decoders, and control word decoder. See figure 2.1.

A single-chip microprocessor is a silicon chip containing all (and only) the parts of a CPU. The chip must include all the parts mentioned above (clock-phase generators, bus controller, processor controller, and execution unit) to be a single-chip microprocessor. Otherwise, it is one chip of a multichip microprocessor. Figure 2.1 is a block diagram of a microcoded implementation of a single-chip microprocessor. Figure 2.2 is a block diagram of a PLA implementation of a

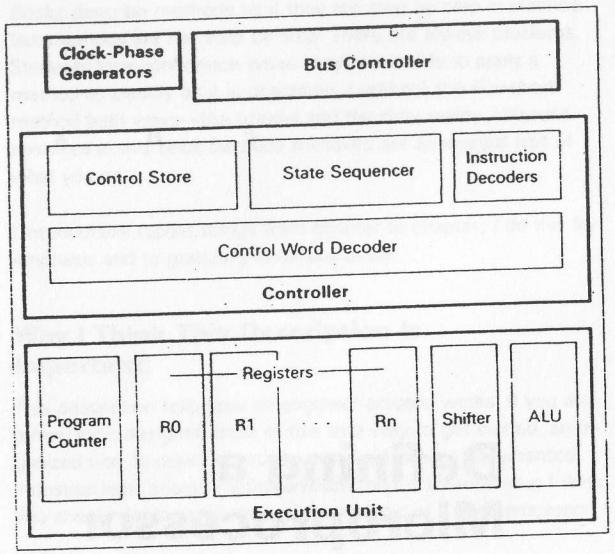


Figure 2.1 Microprocessor (microcoded implementation)

single-chip microprocessor. Figure 2.3 is a block diagram of a random logic implementation of a single-chip microprocessor. From here on, when I use the term "microprocessor," I mean single-chip microprocessor unless I say otherwise. Since I prefer microcoded implementations, I will use those as examples from now on.

"But," you say, "I want to design a high-performance microprocessor, so I want to know how to do a random logic implementation. Your book will do me no good." Our technical folklore says that random logic implementation is faster. That is not necessarily so. If the random logic implementation is faster, it is not because it is done in random logic. Figures 2.1, 2.2, and 2.3, for example, have the same execution unit (exactly what I expect if they implement the same architecture). Where is the critical path? Suppose it's in the execution unit. (A common critical path in an execution unit is the path from a register, through the arithmetic and logic unit [ALU], and into an ALU condition code register.) If so, all three implementations perform equally.

There are many microprocessors commercially available today. You can't tell which are microcoded, PLA, and random logic im-

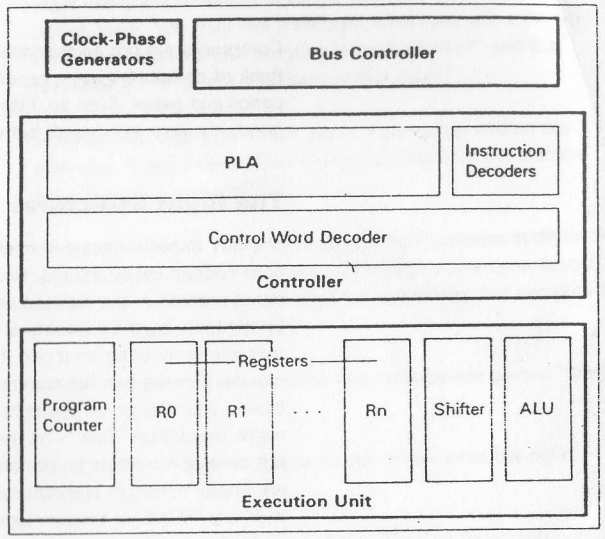


Figure 2.2 Microprocessor (PLA implementation)

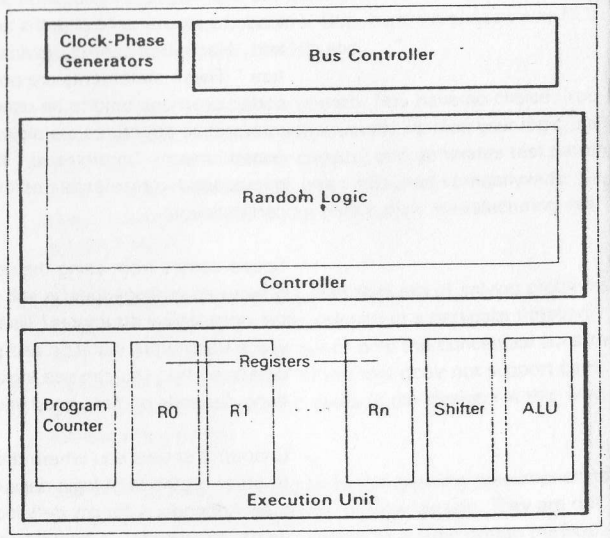


Figure 2.3 Microprocessor (random logic implementation)

plementations based on their performance. Differences in architecture swamp differences in implementation (and make comparisons unsound). The Motorola MC68000 family, which are microcoded designs, are among the fastest microprocessors available.

Why am I harping on this? Because I think that what the implementation looks like (microcoded, PLA, or random logic) when it is done is more nearly related to design method than to performance (or any other input constraint). What you do is start with a specification and design a computer to some goal—size, cost, performance. Don't worry about the form of the implementation (yet). I'll show you how to develop your own design method. Begin by building your method from the things that matter—the specification and the goals. I'll show you how to build a fast microprocessor (or a cheap one or something in-between), and I'll do it with microcoded examples.

Microprocessor Operations Overview

This is an operational overview of the microcoded microprocessor in figure 2.4. Instruction decoders look at the instruction bit pattern to decide which control word sequence in the control store is appropriate. The instruction decoders send the address of the control word sequence to the control store. The control store contains the control word sequences for all the instructions. The state sequencer steps the control store through each control word in the sequence for the instruction. The control word decoder transforms each of the control words into specific control signals for each execution unit element. The execution unit contains the resources for holding and manipulating data. Execution unit pieces (elements) are connected by one or more common internal buses. Transfers between execution unit elements are controlled by the control words. Transfers between the microprocessor internals and the external world (the world beyond the pads) are controlled by the bus controller. (There is a simple connection from the Data In Out register to bus transceivers connected directly to the pads. Similarly, the Address Out [AO] buffer in figure 2.4 goes to drivers connected directly to the pads.) The bus controller responds to commands imbedded in the control words. It runs the external bus protocols that result in instruction fetches (for the instruction decoders) and in operand loads and stores (for the execution unit).

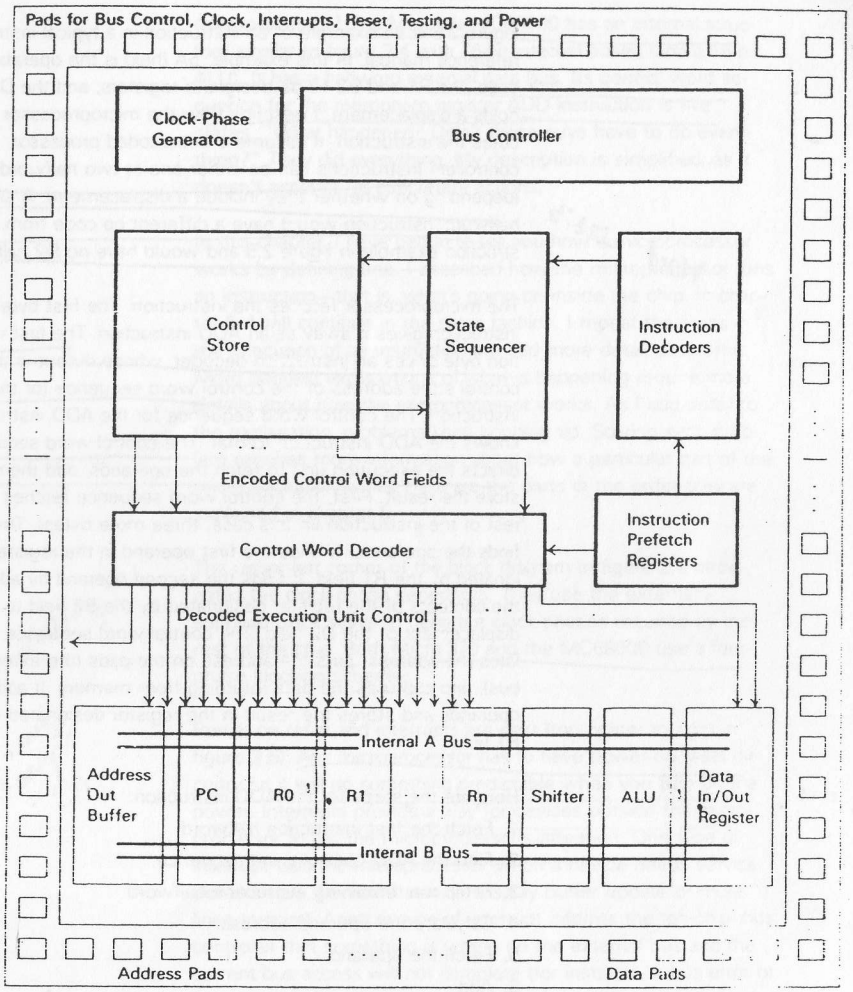


Figure 2.4 Microprocessor chip, with more detail (microcoded implementation)

Instruction

Figure 2.5 is an example of an instruction in a typical instruction reference manual. In this example, 5A (hex) is the operation (op) code, the R1 and B2 fields designate registers, and the D2 field holds a displacement. I describe how the microprocessor executes the instruction. I assume a microcoded processor controller. Instructions can be either one or two halfwords long (depending on whether they include a displacement). A single halfword instruction would have a different op code from the instruction example in figure 2.5 and would have no D2 halfword.

The microprocessor fetches the instruction. The first byte of the instruction gives it away as an ADD instruction. The first instruction byte drives an instruction decoder, whose output is the control store address of the control word sequence for the ADD instruction. The control word sequence for the ADD instruction knows the ADD instruction format. The control word sequence directs the execution unit to fetch the operands, add them, and store the result. First, the control word sequence fetches the rest of the instruction (in this case, three more bytes). Then it finds the operands. It finds the first operand in the register designated by the R1 field. It finds the second operand by adding the contents of the register designated by the B2 field to the displacement of the D2 field. The control word sequence calculates the address, puts the address on the pads (the external bus), and captures the data returning from memory. It adds the operands and stores the result in the register designated by the R1 field.

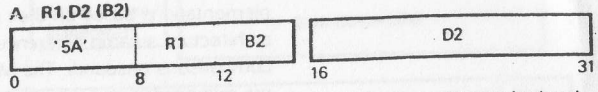
Here are the steps for the ADD instruction:

1. Fetch the first instruction halfword.
2. Find the ADD control word sequence.
3. Fetch the remaining instruction halfword.
4. Calculate the operand address.
5. Fetch the operand.
6. Add.
7. Store the answer.

It isn't quite that simple. This works for one instruction, but you must be able to execute a program (sequence of instructions). How do you get to the next instruction? How did you get here from the last one? The processor controller does this. One way to execute a sequence of instructions is to have the current instruction fetch and decode the next instruction. In a micro-

ADD

A R1,D2 (B2)



The second operand is added to the first operand, and the sum is placed in the first-operand location. The operands and the sum are treated as 16-bit signed binary integers. The first operand is in the register specified by the R1 field. The second operand is in memory. The address of the second operand is formed by adding the displacement specified by the D2 field to the contents of the base register specified by the B2 field. An overflow causes a program interruption when the fixed-point overflow mask bit is 1.

Resulting Condition Code	Program Exceptions
0 Sum is zero	Access (fetch)
1 Sum is less than zero	Fixed-point overflow
2 Sum is greater than zero	
3 Overflow	

Figure 2.5 The ADD instruction

programmed controller, this is necessary to find the location of the control word sequence to execute the next instruction.

Assume that you have just begun execution of the ADD instruction. Here are the steps for the instruction:

1. Fetch the remaining instruction halfword.
2. Calculate the operand address.
3. Fetch the operand.
4. Add.
5. Store the answer.
6. Update the program counter (PC).
7. Fetch the first halfword of the next instruction.
8. Find the address of the next instruction's control word sequence.
9. Branch to the next instruction's control word sequence.

The steps in this sequence have been renumbered from the previous list of steps to reflect a change in instruction execution strategy. The first two steps of the initial sequence became the last four steps of the current sequence. Instead of each instruction being an independent sequence, as it is in the first set of steps, each instruction connects to the next instruction by doing its fetch and decode. These steps can execute a stream of ADD instructions. If you have a series of ADD instructions, you would execute the above steps multiple times. The first five steps do

the ADD instruction, and the last four steps connect it to the instruction stream.

A "step" is not a control word (this isn't really the control word sequence); it's only what the control word sequence must do. The control word sequence defines a series of states, and it may take several states to do each of the steps for the ADD instruction. (In a microcoded implementation, a state corresponds to one control word.)

Assume that the microprocessor in figure 2.4 has a 16-bit external data bus and 16-bit internal data buses (along with a 16-bit ALU). The following steps execute the ADD instruction:

1. Fetch the remaining instruction halfwords.
   One state to fetch the second halfword of the ADD instruction.
2. Calculate the operand address.
   One state to add the D2 displacement and the contents of the B2 register.
3. Fetch the operand.
   One state to fetch the data halfword (put the address on the pads and wait for the operand halfword).
4. Add.
   One state to add the operands.
5. Store the answer.
   One state to store the result in the R1 register.
6. Update the PC
   One state to increment the PC.
   One state to save the incremented value.
7. Fetch the first halfword of the next instruction.
   One state to put the PC value on the pads and wait for the first half of the next instruction.
8. Find the address of the next instruction's control word sequence.
   One state to put the next instruction into the instruction decoder.
9. Branch to the next instruction's control word sequence.
   Zero states—this step is accomplished as a part of the previous step.

For a halfword (16-bit) external data bus and halfword (16-bit) internal buses, the sequence is nine states. How does this compare with the execution time (in states) for a commercial

microprocessor? The Motorola MC68000 has an internal structure similar to figure 2.4 with 16-bit internal buses (and a 16-bit ALU). It has a halfword external data bus. Its control word sequence for the memory-to-register ADD instruction is five states. "What happened? Didn't those guys have to do everything?" They did everything. My description is simplified, as it doesn't account for concurrent actions.

In this chapter, I have begun to tell you how a microprocessor works by defining one. I described how the microprocessor runs an instruction—that is, what's going on inside the chip. In chapter 5, I will continue in the same fashion. I repeat the steps in the execution of an instruction but add more detail each time. More detailed explanations of what is happening require more details about how the microprocessor works. As I add detail to the explanation, problems keep popping up. Solving each problem requires more information about how a particular part of the microprocessor works. Here are the parts in the order they are explained in chapter 5.

The upper left corner of the block diagram in figure 2.6 designates the clock-phase generators. They use the externally supplied clock signal to generate clock phases required by the rest of the chip. Both Micro 370 and the MC68000 use a four-phase clocking scheme.

Power-on reset and interrupts are next (top center to right in figure 2.6). Any microprocessor has to have power-on reset circuitry so it will do something predictable when you turn on the power. Interrupts provide a way for devices outside the microprocessor to get the microprocessor's attention. One type of interrupt tells the microprocessor when a device needs service (for example, keyboard service, display buffer update or more lines to print). Another type of interrupt informs the (on-chip) bus controller that something is wrong on the external bus and the current bus access will not complete (for instance, a bus error or page fault).

Interrupts lead naturally to the next state control (a part of the processor controller's state sequencer). Normally, the microprocessor is just running a user's program. An interrupt comes in and changes what the processor controller does (usually at an instruction boundary). If there is no interrupt, the next state control selects control store addresses from the output of the control store, the branch control unit, or the instruction decoder.

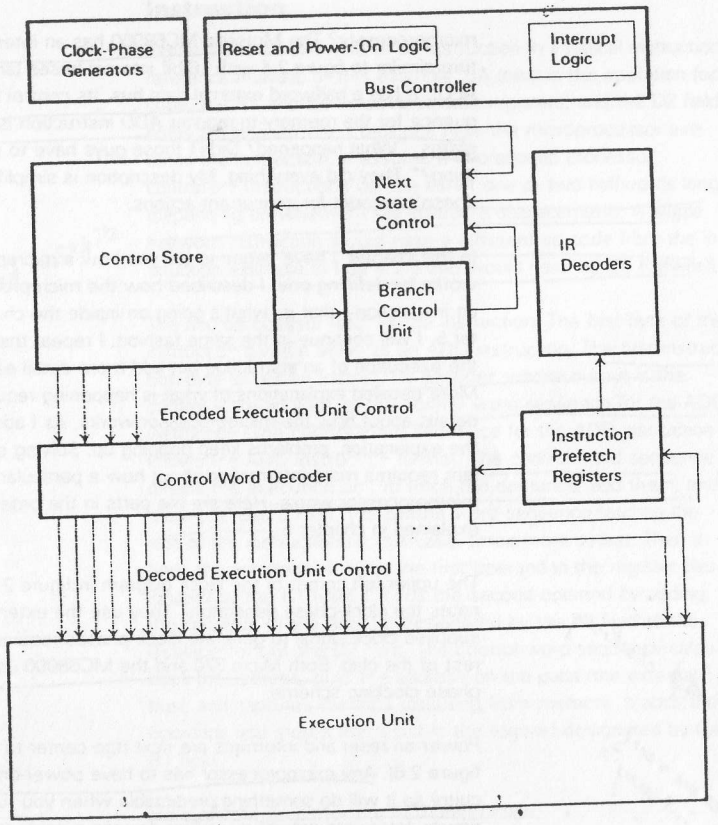


Figure 2.6 Microprocessor block diagram

The branch control unit provides the means for decision-making in the microcode. Both Micro/370 and the MC68000 provide four-way branches using a partial address from the control store and altering some of the control store next address bits based on conditions in the execution unit.

The control store holds the control words. Part of each control word is decoded to control the execution unit elements, and part helps run the state sequencer (by saying where to get the next control store address or even by providing it).

The control word decoder translates the (compact) control word into the exact lines needed to control each execution unit element. The control word decoder also mixes the information from the control store with information from the instruction register and with timing information. In the MC68000, the part of the control word that helps run the execution unit is 66 bits wide. It is decoded into about 180 bits to run the execution unit control points. In Micro/370, the part of the control word that helps run the execution unit is 71 bits wide. It is decoded into about 300 bits to run the execution unit control points.

An execution unit control point is a single control line leading to a macro in the execution unit. For example, a single control line might gate the value of a single register onto a bus. The load signal for a register would be another single control line. The op code control for the ALU might be four control lines. Each of these lines becomes a control point entering the execution unit from the processor controller.

The instruction prefetch registers allow the microprocessor to overlap the execution of the current instruction with decode of the next instruction and with the prefetch of the halfword after the next instruction. There are three registers. One holds the currently executing instruction and is used by the control word decoder. Another holds the next instruction and drives the instruction decoder. The last register receives the halfword following the next instruction, when the halfword arrives from the external bus.

The bus controller runs the electrical protocol to communicate with the outside world. The bus controller detects and synchronizes external interrupts, runs memory access cycles, and arbitrates control of the external bus.

This is a preview of the explanation of how a microprocessor works, presented in chapter 5.

CHAPTER

3

Hardware Flowcharts

The Flowchart Method is the procedure and notation I use to design the CPU of a computer. The method works for general-purpose and for special-purpose CPUs.

A CPU has a "controller" and an "execution unit." The execution unit is a collection of fast but latent capabilities (registers, ALUs, shifters, and data paths). The controller controls the execution unit by telling the execution unit what to do when. The controller determines the CPU's "personality."

Designs often begin with an appeal: "We need a CPU that's twice as good as any rival's." Computer architects turn the appeal into an English description of the machine (in IBM's System/370, this is the *Principles of Operation* manual, form no. GA22-7000). Engineers implement from the English description, using logic design and circuit design methods. We have lots of books to help us with logic design and circuit design, but nobody says how to transform the English description into the kind of formal description circuit designers need.

It's much like a mathematical word problem. The hard part is getting the equations from the written description of the

problem. Once you have the equations, you can apply documented methods to find the solution. The English description of a chip is like a book-length mathematical word problem. Hardware flowcharts are a bridge between English and the logic designer; they are a compact formal description of what the CPU does.

The method I describe was used to design the controller for the Motorola MC68000 and IBM Micro/370 microprocessors. 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 controller. (Doing this imposes a structure on the English specification; the problem is to *find* an efficient structure.) The block diagram is one of the procedure's *outputs*.

Flowcharts show the design as the flow of simple actions. An example 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 statements is called a task; states can be one (really zero) or more tasks. I depict the flow of states by boxes (one for each state); I draw these in a specific format, and it is important 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 (flavors of bipolar, nMOS, CMOS). Decisions in the procedure are based on the capabilities of the particular technology. The procedure does not depend on the implementation method. This means that the same flowcharts are used to implement the chip with combinational logic, PLAs, or microcode. In chapter 4, I show how to implement a simple microprocessor using flowcha

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' worth of computers and fancy equipment.

2. Design automation should be subservient to the design method. It should support the design procedure, 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 how to get from the architecture to the implementation. They link the programmer's (external) model and the hardware (internal) implementation. Flowcharts specify exactly how commands from the instruction set are carried out using execution unit hardware. You must have the instruction set summary and an execution unit specification before you 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 User's Manual* for the Motorola MC68000 or the *iAPX Book* for the Intel 8088.) The instruction set summary describes:

1. Instruction formats
2. Operations (ADD, AND, SUB, and so on)
3. Addressing modes (Base Plus Displacement, Register Indirect, Indexed, and so on)
4. Registers (as seen by the programmer)

Execution Unit

A microprocessor's execution unit (or data path) details are not usually published for several reasons: Users do not want to know, users should not know, or manufacturers want competitive advantages kept secret. You need a block diagram of the execution unit that shows the following:

1. Programmer's register set
2. Additional registers (such as the instruction register, program counter, and temporary registers)
3. ALU and any special function units (such as a shifter)
4. Internal data paths
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

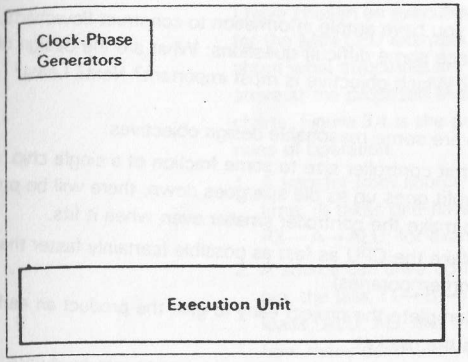
pieces (registers, buses, arithmetic units, and so on). The rules of operation also tell clock phases, timing, and electrical load constraints for the pieces. These rules are imposed by circuit design limits.

If you are responsible for the flowcharts, you should do the execution unit first. To design the execution unit, I recommend doing trial flowcharts for ten frequently used instructions to determine an initial execution unit structure. I think a simple bus-oriented structure is best, so I start with that. In a current (1987) very large scale integration (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.

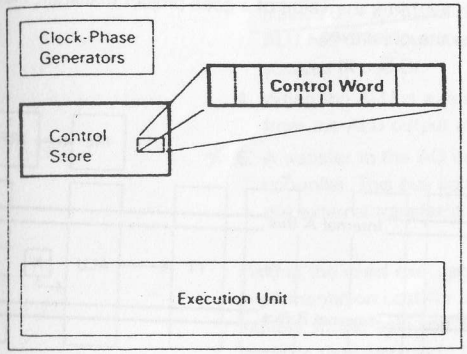
The execution unit will evolve. I proposed the initial execution unit for Micro 370 in January 1981. It went through about twenty-three major revisions before I completed the flowcharts. These changes are expected—and are supposed to happen. For example, in writing flowcharts for the instructions, you find an instruction you cannot implement efficiently. You can't do a Booth's algorithm multiply efficiently because you can't "see" the low-order bits in the multiplier. Since the multiplier normally resides in the shifter, you just wire the low-order shifter bits to the branch control unit. Perhaps you need a special direct path from the ALU to the Data Temporary register (DT). You can just move the DT next to the ALU and wire the direct path. If you need something, add it. The circuit designers will tell you when you're not being reasonable.

Illustrated Flowchart Method Overview

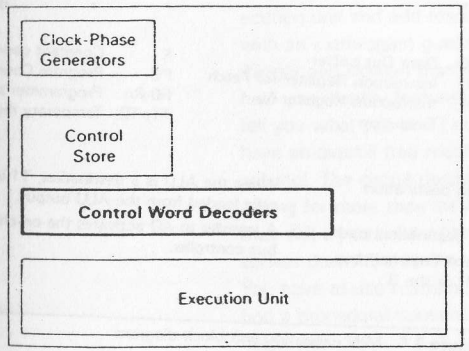
Figure 3.1 shows the development of the implementation using the flowchart method. To avoid confusing details, I illustrate the method with a simple microprocessor, called MIN. Figure 3.2 shows the instruction format and register set; figure 3.3 shows part of the instruction set summary. This subset is adequate to demonstrate flowchart construction. Figure 3.4 shows a sufficiently detailed block diagram of the execution unit. It also includes some rules of operation; others will be added as I progress.



- The architecture specification is the only input.
- Begin with a guess for the execution unit.
- Do flowcharts for the instructions.
- This modifies and refines the execution unit and develops the control store and control strategy.
- The final execution unit is derived output.

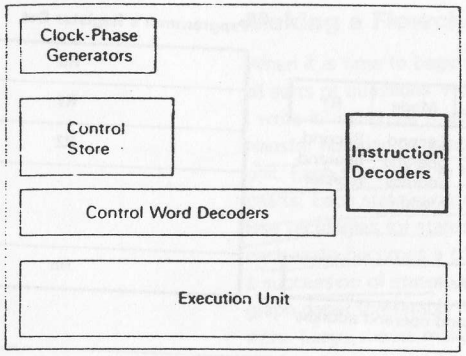


- Once the flowcharts are fairly complete, derive the control word format using the flowchart states.
- When the flowcharts are complete, so is the execution unit.
- Control word format is derived output.

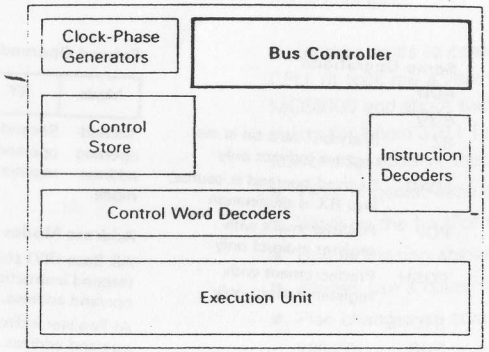


- After defining the control word format, you assign bit patterns to the control fields in a way that minimizes control word decoders between the control store and the execution unit.

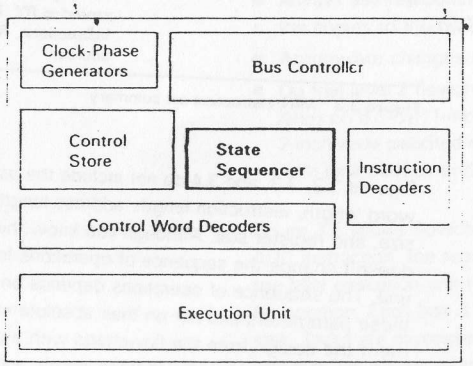
Figure 3.1 Development of implementation using the flowchart method



- Instruction decoders are defined by the flowcharts and the architecture specification.



- Completed flowcharts, control word format, and the initial bus specification define the bus controller.



- Last is the logic of the state sequencer, the part of the chip that says what to do next. ("Where's the next control word?")
- Once everything around it is defined, you build exactly what you need! (The state sequencer is derived output.)

Figure 3.1 (continued)

Instruction Format

First Word

OP	RX	Mode	RY
Operation code	First operand register	Second operand address mode	Second operand register

Second Word

Displacement

Optional, depending on second operand address mode

Programmer's Register Set

R0
R1
R2
.
.
Rn

Figure 3.2 MIN instruction format and register set

Some Operations

ADD
AND
BZ Branch if zero bit is set, register indirect only
LOAD Second operand is source, and RX is destination
POP Postincrement with register indirect only
PUSH Predecrement with register indirect only
STORE
SUB
TEST

Second Operand Address

Mode	RY
Second operand address mode	Second operand register

Address Modes

AB-Base (RY) plus displacement (second instruction word) 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 also is an operand source.

Figure 3.3 MIN instruction set summary

Figures 3.2, 3.3, and 3.4 do not include the usual details about word length, instruction length, address length, bus width, ALU size, and register size. Although you know this information, it doesn't change the sequence of operations for the execution unit. The sequence of operations depends on relative values of these parameters and not on their absolute values. You implement the design from the flowcharts with a particular word length, instruction length, address length, and so on. Don't clutter your flowcharts (or your notation) with details you don't need.

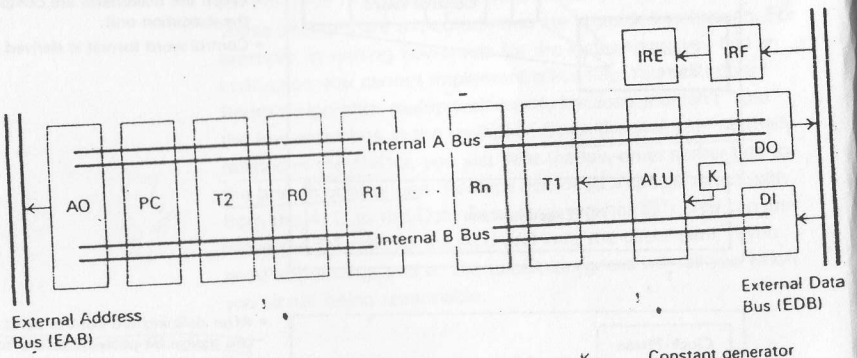
Flowchart Objectives

Now you have ample information to construct flowcharts, but you face some difficult questions: What are the design objectives? Which objective is most important? 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 CPU as fast as possible (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 or she ranks the objectives.



ALU Arithmetic and Logic Unit
AO Address Out buffer
DI Data Input register
DO Data Out buffer
IRF Instruction Register for Fetch
IRE Instruction Register for Execution
K Constant generator
PC Program Counter
R0-Rn Programmer's registers
T1, T2 Temporary registers

Example Rules of Operation

1. A transfer from source to bus to destination takes one state time.
2. A source can drive up to three destination loads.
3. Inputs to the ALU are from the A (internal) bus and either K (values 0, +1, −1) or the B (internal) bus.
4. When the ALU is a destination, T1 is automatically loaded from the ALU output.
5. A transfer to AO activates the on-chip external bus controller.

Figure 3.4 MIN execution unit block diagram

I have chosen an execution unit with a simple two-bus structure for the MIN CPU example. You talk to the circuit designers about what structures are reasonable for the technology. You arrive at the proposed execution unit by doing some trial flowcharts. Figure 3.4 is the proposed execution unit. Here are some rules of operation:

1. A transfer from source to bus to destination takes one state time. (It takes one flowchart state to execute the task RX→A→ALU, for example.)
2. A source can drive up to three destination loads. (For example, the task T1→B→ALU,AO,PC has three destination loads: ALU, AO, and PC.) The circuit designer will tell you how many destination loads each source can drive.
3. Inputs to the ALU are from the A (internal) bus and either K (values 0, +1, −1) or the B (internal) bus. One side of the ALU has one input source (A) and the other has two input sources (K and B).
4. When the ALU is a destination, T1 is automatically loaded from the ALU output at the end of the state time.
5. A transfer to the AO buffer activates the on-chip external bus controller. This bus controller postpones the next state until the external transfer is complete.

Picking the initial execution unit requires some knowledge of implementation cost for the technology you use. The circuit designers should help you with this. It is much better to start with too little than too much. (It's easier to add things than to figure out whether you can throw them away.) Start with a simple execution unit and add resources as you need them. If you begin with an extravagant guess, you may build something fancier (bigger and slower) than you need. The flowchart method can help you identify features that improve performance. It will not tell you what you don't need. And it will not tell you when you have an overkill (too much hardware for the problem you are solving). The circuit designers should warn you when you are asking for more than they can do. If they trust you, however, they will try to build what you want—even if it is too much. It all comes down to this: You are an engineer (the logic designer). You have to use restraint, common sense, and judgment. I can't find a procedural substitute for *you*. I can only tell you what helps me.

Making a Flowchart

When it is time to begin the flowcharts, you will be plagued with all sorts of questions. How do I begin? What do I write? How do I write it? I suggest methods that work for me. 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 comprises one (really zero) or more tasks. Use rectangles for states. (In a microcoded implementation, each state becomes a control word.) A control word sequence is a succession of states. Work on large sheets of high-quality graph paper (preferably 17-by-22-inch vellum with ten 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 can take years to complete the flowcharts for a complicated CPU. (It took me a year to complete flowcharts for the Motorola MC68000 and about two and a half years to complete flowcharts for Micro/370.) To avoid copying several generations of flowcharts, observe these rules:

- Work in pencil. (Use a .5mm 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 reduce the copies to 8½ by 11 inches for easier use.)
- Accumulate changes (in red ink) on a reproduction.
- Do trial level 2 flowcharts (level 2 flowcharts are explained later) on 8½-by-11-inch scratch paper with 1½-inch-high, 2-inch-wide penciled-in rectangles as guides. (I load the copier with junk memos and copy the grid on the back.)

Figure 3.5 shows flowchart sequences for the register-to-register ADD instruction, the register-to-memory ADD instruction using the MIN execution unit (figure 3.4), 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

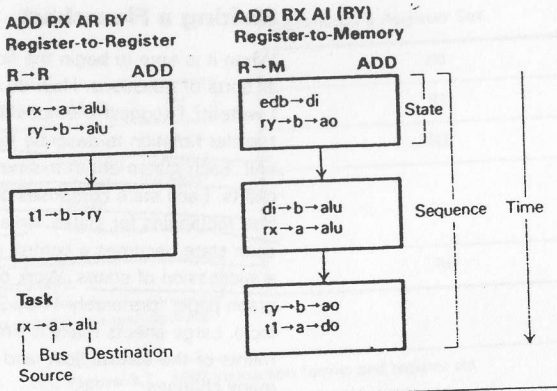


Figure 3.5 Execution of register-to-register ADD and register-to-memory ADD instructions (partial description—operation tasks only)

in each state by source (if there are multiple destinations on a single line, alphabetize them, too); you will use this to compact the flowcharts later on.

In figure 3.5, time advances from the top of the page to the bottom of the page, except within a state. Within a state, tasks appear to be concurrent but are governed by rules-of-operation timing. In a microcoded implementation, each state is one microcycle (and may have phases such as source, transfer, destination, and precharge).

In the register-to-register example (the left flowchart in figure 3.5), I transfer both operands to the ALU in the first state. The output of the ALU is saved in T1 any time there is an ALU operation. In the second state, the result is sent from T1 to RY. Look at the register-to-memory ADD example (the right flowchart in figure 3.5). The first state fetches the memory operand; the second state adds the operands; the third state sends the result to memory.

Something doesn't look right. In the first state, DI is loaded from the external data bus (EDB), but RY is sent to AO after this happens. Wrong! I consider these tasks concurrent (with some implicit timing). They are in alphabetical order. Sending RY to AO initiates the external bus activity that results in the DI transfer from EDB. The tasks are listed in the same state because as far as the state sequencer is concerned, they happen at the same time.

In a microcoded controller, the control word specifies the tasks in a single state. The tasks are commands to the external bus controller, the execution unit, and the state sequencer. Whatever timing is added later, the commands all come out of the control store at the same time. In the case of a read, the transfer to AO initiates the external bus cycle. If the external bus is synchronous, then DI must be valid by the end of the state time. If the bus is asynchronous, the state sequencer "hangs" in the current state until the transfer to DI is completed. In the case of a write, the address and data are transferred to the external address bus (EAB) and EDB, respectively. The state sequencer "hangs" until the external bus controller signals the state sequencer that the external transfer is completed.

There is no explicit notation for transfer from AO to EAB or from the Data Output register (DO) to EDB (or for memory to EDB). I have elected to let them be implied by the context. I view AO and DO as amplifiers (not registers). Because AO is not a register, it does not remember the address between state 1 and state 3 of the register-to-memory sequence (the right flowchart in figure 3.5). Transfers from EDB to the execution unit are not implicit because they can be to the instruction register for fetch (IRF), DI, or both.

Notation

Keep the register transfer notation simple. It must capture the essence of what the CPU is doing without all the details. You may think this is a simple notation invented for just this one case. Well, that's somewhat true. I modify the notation to fit the problem. I want the notation to be a simple, natural, readable way to express what the CPU is doing. That is why the notation is not formally defined. In a formal notation, constructs might prevent natural expression of tasks and hinder the design.

Flowcharts are graphic notations that depict the CPU in two ways:

1. Flowcharts visually emphasize changes in sequence and concurrency for whatever the controller is doing. You see branching and merging in the flow of control. You see how the address calculation sequences and operation sequences are shared. You see all the instructions sharing one common set of address calculation sequences. You see ten instructions sharing the standard dual operand execution sequence

(as the MC68000 register-to-register operand execution sequence does, for example). You see which instructions have an execution sequence all to themselves (multiply or divide in Micro/370, for example).

2. Flowcharts visually communicate the relationship of sequence to concurrency for whatever the controller is doing. You see exactly what is concurrent (tasks) and what is sequential (states), and you see how they are related.

Flowcharts show sequential state flows made up of concurrent tasks. Each task is a sequential source-bus-destination flow. Flowcharts are a flow-intensive notation showing you the concurrent and sequential nature of operations.

Execution Speed

The flowchart sequences in figure 3.5 are incomplete. They do not include the instruction fetch and the 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? How about this definition: The most efficient controller executes a given instruction with the least number of states.

"That's kind of a truism. Give me something I can use—that tells me what to do." You are designing something (a microprocessor) that will be part of a larger system (a board, a personal computer, an instrument). What limits system performance? Is it always your part? Sometimes your part? If your part is the system bottleneck, you did not design it very well. If your part is never the bottleneck, perhaps you spent too much on hardware. The best engineering design achieves the effect of infinite resources (never the bottleneck) at minimum cost. Microprocessor design is a good example. I believe that useful external bus activity in every state is evidence of sufficient controller efficiency. Therefore, I use the following definition for controller efficiency: The controller is efficient if execution never delays external bus cycles. (If some other part of the system is the bottleneck, the controller design is good enough.)

Measuring the microprocessor performance at the pads will not reveal whether you implemented a Cray supercomputer or a controller barely sufficient to make external bus transactions the bottleneck. This is not a measure of bus efficiency or system efficiency; it is a measure of how well you do the controller design.

I can't give a useful general definition for the fastest controller because it depends on what the controller does. I have given a definition that works for a microprocessor, but an applications engineer would not use this definition because he would not want the external bus tied up by the CPU all the time.

Figure 3.6 improves the examples in figure 3.5 with the PC increment and instruction fetch. I removed the lines connecting boxes because they are unnecessary and doing so 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 I do about it? In some states of each flowchart sequence, the major internal buses (A and B) are not both occupied. That's 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.

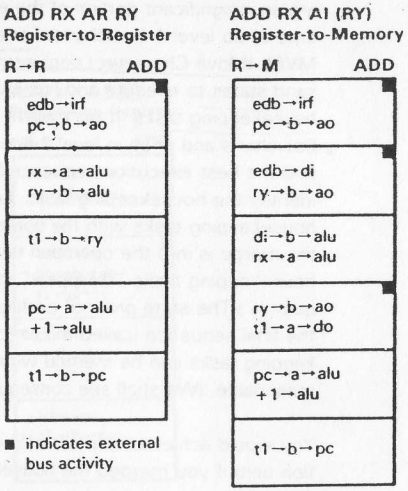


Figure 3.6 Revised execution of ADD instruction examples

Level 1 Flowcharts

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 3.5 shows the operation tasks for two types of ADD instructions. Housekeeping tasks, such as PC increment and next instrument fetch, are common to all instructions. You have some leeway in deciding when these tasks are accomplished. The tasks are essentially independent for all instructions, so you should treat them separately (initially). Separate kinds of tasks so you can optimize the execution of the operation tasks.

Figure 3.7 shows the flowcharts in a format designed to aid later 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. Do level 1 flowcharts for most of the instructions and then begin level 2 flowcharts. You do not have to do level 1 flowcharts for all instructions. If you have instructions for which housekeeping tasks are an insignificant portion of the execution time, it is a waste of time to do level 1 flowcharts. For example, the System/370 MVCL (Move Character Long) instruction may take several thousand states to execute and has only a couple of states of housekeeping tasks. It isn't worth doing twice (once in level 1 flowcharts and once in level 2 flowcharts). Level 1 flowcharts find the best execution sequence for the operation tasks and identify the housekeeping tasks. Level 2 flowcharts merge the housekeeping tasks with the operation tasks. The direction of the merge is into the operation tasks. (You want to make the housekeeping tasks "disappear" into the operation task sequences.) The state 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. (We shall see consequences of this merging later.)

You would achieve the most efficient execution (for this execution unit) if you 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

ADD RX AR RY
Register-to-Register

R→R	ADD
rx→a→alu ry→b→alu	edb→irf pc→b→ao
	pc→a→alu +1→alu
t1→b→ry	irf→ire t1→b→pc
Operation tasks	Housekeeping tasks

ADD RX AI (RY)
Register-to-Memory

R→M	ADD
edb→di ry→b→ao	edb→irf pc→b→ao
di→b→alu rx→a→alu	pc→a→alu +1→alu
ry→b→ao t1→a→do	irf→ire t1→b→pc
Operation tasks	Housekeeping tasks

■ indicates external bus activity

Figure 3.7 Level 1 flowcharts for two types of ADD instruction

states in the final execution sequence be significantly less than the total states in housekeeping task and operation task sequences. Since the microprocessor allows only one external bus access per state, you have done enough when you merge housekeeping tasks into an operation task sequence in a way that produces useful external bus activity (including housekeeping accesses) in every state.

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

Be careful merging housekeeping tasks into operation task sequences. You would not want to increment the PC before you computed a PC-relative branch address. Merging housekeeping tasks into operation task sequences is challenging and fun because it requires skill and care. You may reorder tasks, change the execution unit, and try dozens of combinations and sequences to get the most efficient execution sequence. This is design. You are working to find the best execution unit for the instruction set and the best controller for the execution unit. If you like puzzles, it won't even seem like work. This is how you are creating the controller. I will show (later) how assumptions you make in the flowcharts translate to hardware in the controller. You get the controller you need, which is better than choosing a controller and trying to make it do what you need.

I added one more thing in figure 3.7. IRE is the instruction register for execution (see figure 3.4). It allows a rudimentary prefetch. IRE holds the current instruction and drives the register selection decoders (for RX and RY). IRE is loaded at the beginning of a state, and decoding will be stable within one state time. It must not be changed until after the last RX or RY reference in the flowchart sequence for the current instruction. Each instruction (sequence of operation and housekeeping tasks) is associated with a particular register pair (RX, RY) established by IRE. The instruction register for fetch (IRF) can be used to hold the next instruction until the current instruction is done. It can be loaded anytime during the current instruction—this is the simple prefetch. More accurately, IRF gets the word following the current instruction. (It may not be the next instruction if the current instruction is a branch or a two-word instruction.)

Level 2 Flowcharts

Figure 3.8 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. (You could do better with a more complicated execution unit and controller.) Register T2 saves the operand address in the register-to-

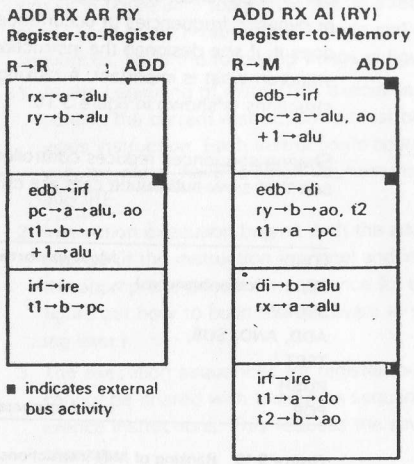


Figure 3.8 Experimental reduction of the level 1 flowcharts

memory ADD example. Because T2 contains the memory address (for the second operand) and the static decoders (which are driven by the IRE) are available, there are no more RX or RY references. The last state can change IRE and store the result.

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, you will discover inefficiencies in the structure of the execution unit as you 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 3.4), I would not have been able to do the instruction in fewer than 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 3.9 shows a register-to-register ADD sequence for an execution unit with no path from the A bus to the AO buffer. Beware! The increased complexity of the execution unit can increase the number of unique states and result in a larger controller. Increasing the complexity of the execution unit implies more execution unit hardware, too. Only after carefully studying the flowcharts and the execution unit would I suggest execution unit changes to improve the efficiency of the overall design.

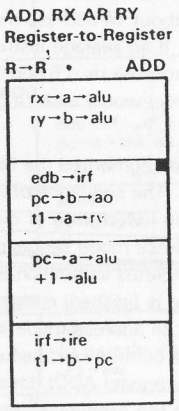


Figure 3.9 ADD sequence for an execution unit in which AO is connected only to the B (internal) bus

The flowchart method helps design hardware. I believe that it is a good, workable method. It is not a rote recipe for good design; you still have to know something about what you are doing. For example, if you start with a fancy execution unit, there is nothing that tells you to throw away expensive hardware. A little arrow drawn on your execution unit may imply a 32-bit data path (lines, space, power) with pass transistors and control signals. Be sure you need it. You should know the cost of what you ask for. Start with a simple execution unit and add what you need. The circuit designers should tell you when you ask for too much.

Feedback on Controller Design

Use the format in figure 3.7 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. That is too many, however, because I write only one sequence for each instruction—independently of which registers are specified. (This is an advantage of static decoders for the register fields.) I need only decode the op code and the mode bits in the effective address field (see figure 3.3). Suppose the simple MIN CPU has k operations (ADD, AND, OR, SUB, and so on) and a address modes (Register Indirect, Base Plus Displacement, Indexed, and so on). If any address mode is valid for any operation, I would need $k*a$ instruction sequences.

Clearly, this number can be large. For example, the Motorola MC68000 has about 14 address modes and more than 50 instruction types. If an average instruction has 8 states, then I must implement more than 5,600 (50*14*8) states in the controller. Such a chip would make a good office partition.

Note that I have segmented the flowchart sequences for executing instructions. The sequence of flowchart states for an address mode calculation (which may or may not include operand fetch) is called an address mode sequence. The sequence of flowchart states that completes instruction execution (once the address mode sequence is finished) is called an execution sequence. The combination of an address mode sequence and an execution sequence forms a control word sequence. If an instruction (such as a register-to-register ADD) does not need an address mode sequence, then the execution sequence and the control word sequence are the same. I will use this terminology throughout the book.

If most address modes can be used with most operations, why not share address mode sequences? (Address mode sequences calculate the operand address, fetch the operand, and place it in DIN.) ADD Register Indirect and OR Register Indirect, for example, would share a common Register Indirect address mode sequence. Also, the Register Indirect address mode sequence and the Base Plus Displacement address mode sequence could branch to the same execution sequence for the OR (or any other) instruction. The operand will be in DIN; the execution sequence doesn't care how the address was calculated to put it there. If you share the address mode sequences among the execution sequences, you need only $k + a$ sequences, and that is in keeping with the goal to reduce controller size.

This is a good idea, but what will it cost? It's not free. Suppose you 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 (branching to and returning from the address mode sequence), but it lets the controller be much smaller (since the address mode sequences are shared by the execution sequences). The size and speed goals conflict, so a trade-off 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. The designer could base the ranking on static or dynamic frequencies of occurrence. However the designer does it, if she designed the instruction set, she must take the stand on what is important. A ranking for the sample MIN instructions is shown in figure 3.10.

Sharing sequences reduces controller size. From the ranking you see that slow subroutine calls are costly because at least three

LOAD	Most important
BZ, . . . (other branches)	
STORE	
ADD, AND, SUB, . . .	
TEST	
PUSH	
POP	Least important

Figure 3.10 Ranking of MIN instructions

of the four most important instruction types can use any address mode (hence, would have to branch to and return from an address mode sequence). You will not use subroutine calls. You assume that address mode sequences can be shared by initially entering the address mode sequence and branching directly to the appropriate execution sequence. One way to do this in a microcoded controller is to have the instruction decoder provide more than one control store 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. You have not, however, constrained the implementation of the controller to be combinational or microcoded; that choice lies in the future. You do not even have a block diagram of a controller, and you do not want one yet because you want the procedure to give you the requirements for the controller independently of what you think a controller should look like. The flowchart method finds requirements for the controller that best fits what the CPU wants to do (the architecture specification).

Doing Level 1 Flowcharts

The level 1 flowcharts for a subset of MIN instructions are shown in figure 3.11 (pages 37–38). In a real CPU, the flowcharts have many more address mode and execution sequences. Note the following things in figure 3.11:

1. At the beginning of instruction execution, IRE is assumed to contain the current instruction. It must be loaded by the previous instruction. Each instruction's control word sequence will, therefore, have to fetch the next instruction and load it into IRE.
2. Instruction execution begins with the address mode sequence (if the instruction has one) and implicitly branches to the appropriate execution sequence for completion. (We will figure out how to build the hardware to support this branching later.)
3. The execution sequences for register-to-register instructions cannot be shared with execution sequences for memory reference instructions. This reduces the savings from sequence sharing.
4. The execution sequences for standard dual operand instructions (ADD, AND, SUB, and so on) are identical except for the

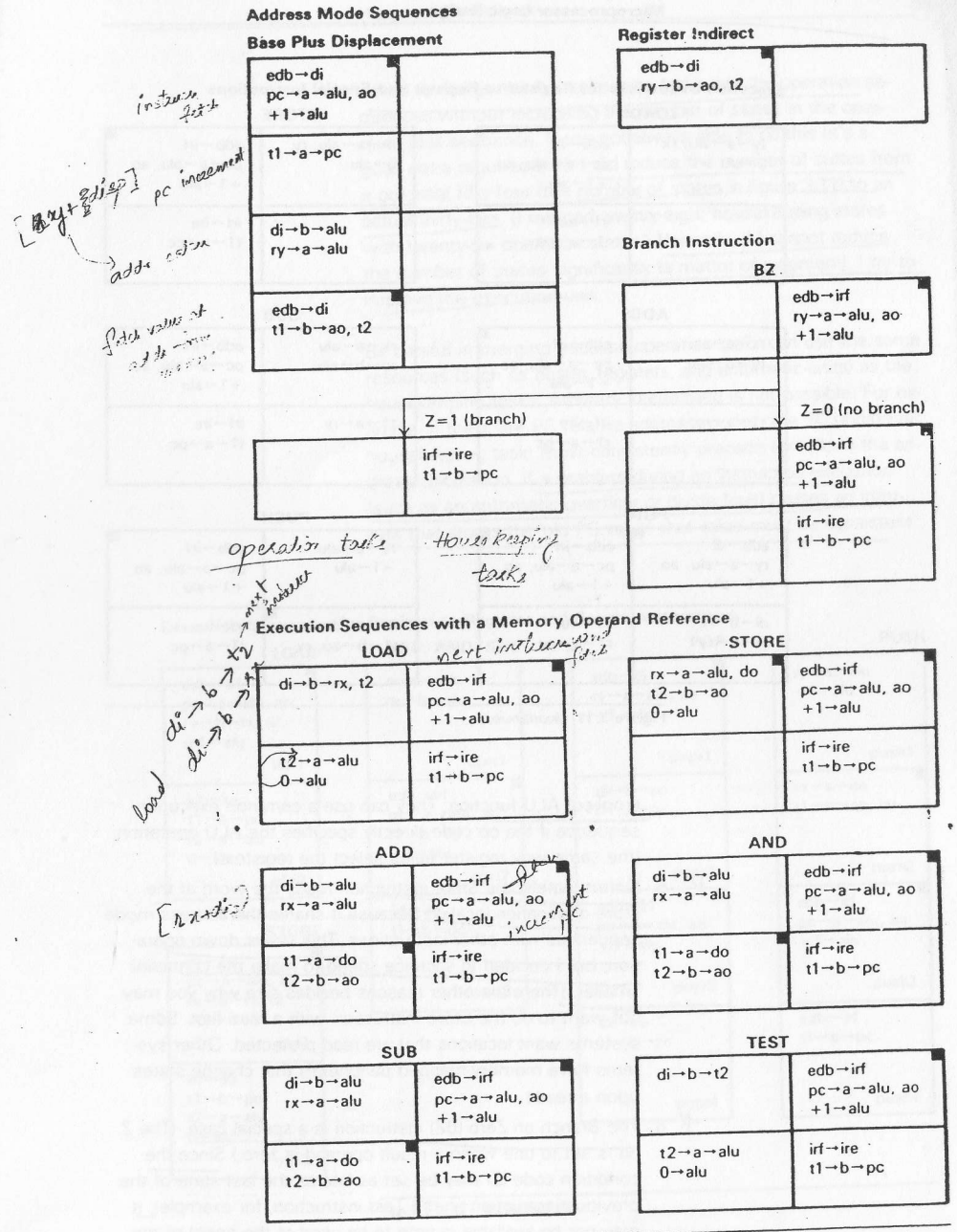


Figure 3.11 Typical level 1 flowcharts for the MIN CPU (continues)

Execution Sequences for Register-to-Register and Special Instructions

LOAD

ry→a→alu, rx 0→alu	edb→irf pc→a→alu, ao +1→alu
	irf→ire t1→a→pc

STORE

rx→a→alu, ry 0→alu	edb→irf pc→a→alu, ao +1→alu
	irf→ire t1→a→pc

ADD

rx→a→alu ry→b→alu	edb→irf pc→a→alu, ao +1→alu
t1→a→ry	irf→ire t1→a→pc

SUB

rx→a→alu ry→b→alu	edb→irf pc→a→alu, ao +1→alu
t1→a→ry	irf→ire t1→a→pc

POP

edb→di ry→a→alu, ao +1→alu	edb→irf pc→a→alu, ao +1→alu
di→b→rx t1→a→ry	irf→ire t1→a→pc

PUSH

ry→a→alu −1→alu	edb→irf pc→a→alu, ao +1→alu
rx→a→do t1→b→ao, ry	irf→ire t1→a→pc

Figure 3.11 (continued)

implied) ALU function. They can use a common execution sequence if the op code directly specifies the ALU operation (the same way register fields select the registers).

5. Unfortunately, the Store instruction reads the word at the store destination location because it shares the address mode sequences with other instructions. This slows down operation, but I decided to sacrifice speed to make the controller smaller. (There are other reasons besides size why you may not want to do the Store instruction with a read first. Some systems want locations that are read protected. Other systems have memory-mapped peripherals that change states upon a read.)
6. The Branch on Zero (BZ) instruction is a special case. (The Z bit is set to one when a result operand is zero.) Since the condition code (Z) may be set as late as the last state of the previous instruction (in the Test instruction, for example), it may not be available in time to be used at the onset of the

next instruction—in this case, BZ. (Because of the simple prefetch, the instruction decoder is operating concurrently with the execution unit. Information that can change in the execution unit cannot, therefore, be used by the instruction decoder.) As a result, use of the condition code must be deferred at least one state time in the instruction sequence. The branch appears between the first and second states of the task sequence for the branch instruction. Because I need a delay state for the condition code to settle, I must decide how to use the state. The example in figure 3.11 (page 37) shows an anticipated branch prefetch (it is discarded if the branch is not taken). An alternative would be to fetch the next sequential instruction and discard it if the branch is taken. The instruction set designer should be able to tell you which alternative to use.

Note that there are no conditional tasks. If you get to a flowchart state, you always execute all the tasks in the state. There are two types of conditional branches, visible and invisible. One conditional branch is explicitly used in the instruction. Examples are BZ (Branch on Zero), BN (Branch if Negative), and BP (Branch if Positive). Another conditional branch is available to the microcode but not visible to someone using the instruction set. If, for example, you have to implement a Multiply instruction but you do not have a hardware multiply unit, you will do the multiplication with a shift and add or subtract algorithm. You will need conditional branches in the microcode to test the multiplier bits and to detect the end of the algorithm.

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

Doing Level 2 Flowcharts

Figure 3.12 shows the housekeeping tasks merged into the operation task sequences for the instructions in figure 3.11. I also integrated the standard dual operand instructions (ADD, AND, SUB, ...) into a single sequence. (This assumes IRE will select the ALU operation in the same way it selects the registers.)

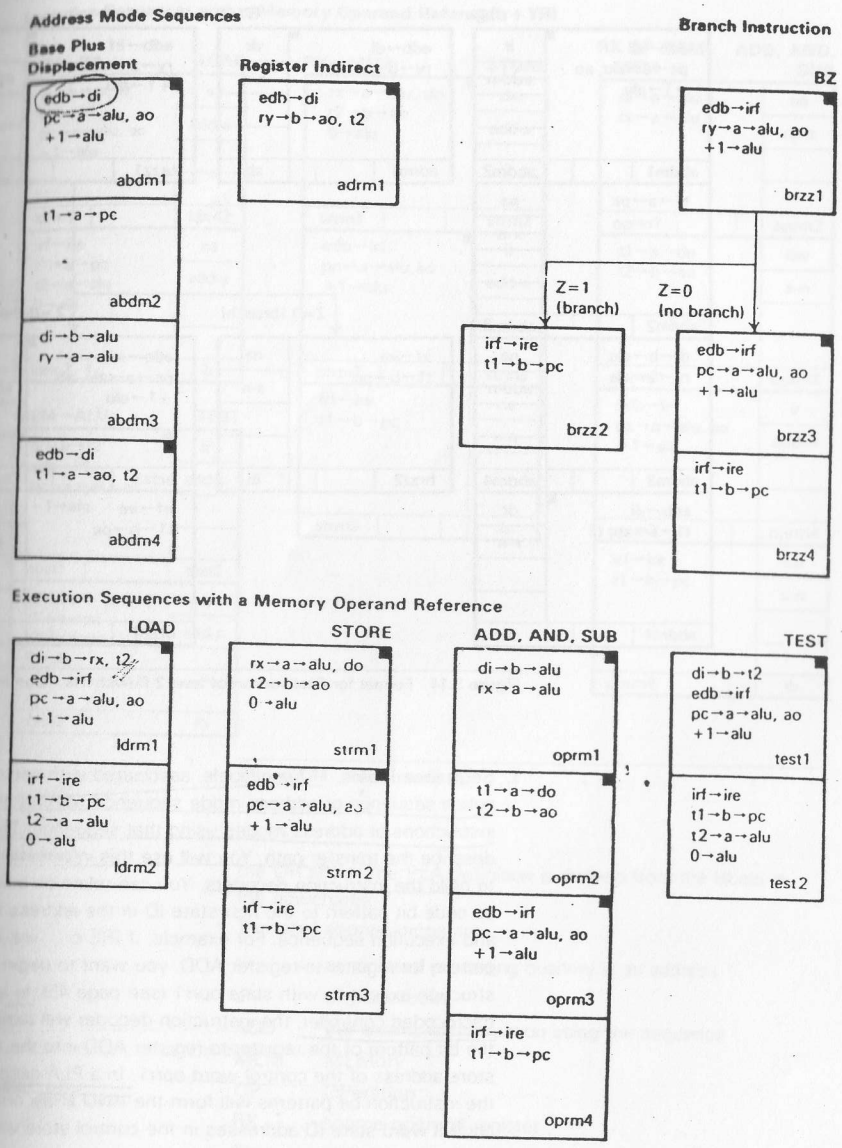


Figure 3.12 Merged level 1 flowcharts for some MIN instructions (continues)

I tried to merge the housekeeping tasks into the operation sequences without increasing the number of states in the operation task sequence. I was not always able to do this (it's a goal, not a requirement). I did reduce the number of states from a potential fifty-four (the number of states in figure 3.11) to an actual thirty-five. (I merged twenty-eight housekeeping states with twenty-six operation states.) Normally, if I cannot reduce the number of states significantly (a matter of judgment), I try to improve the execution unit.

Be careful in merging because operation tasks can use the same resources (such as buses, registers, and arithmetic units) as the housekeeping tasks. Arbitrary interleaving is not possible. For example, if there are PC relative address modes, the PC update (a housekeeping task) must consistently precede (or follow) the address calculation. If a problem during an instruction execution (such as an arithmetic overflow or divide fault) causes an interrupt that stores the old PC value, that value must be consistent

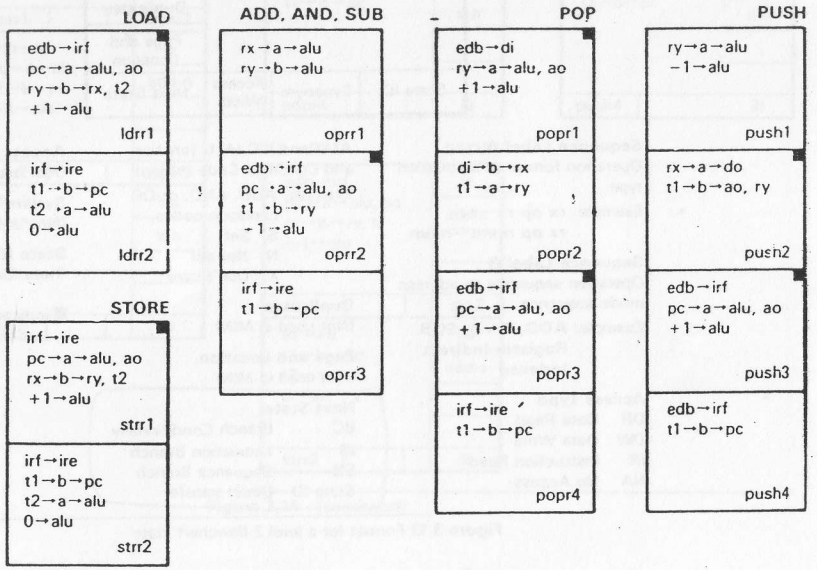
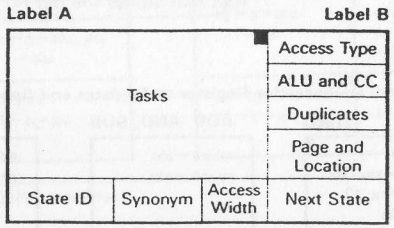


Figure 3.12 (continued)

for all instructions. (The instruction set designer should tell you if you have to store the old PC for interrupts and what the PC should point to.)

Before you can transform the flowchart description to a hardware implementation, you must identify the states. In figure 3.12, I put state identifiers in the lower right-hand corner of each state.

If you add descriptive information, you can ease the transition from flowcharts to hardware, and you make the flowcharts easier to use. But what descriptive information will help? What information do you 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 figures 3.13 and 3.14 (pages 42–45).



Sequence Label A
Operation format or instruction type
Example: **rx op ry→ry**
rx op mem→mem

Sequence Label B
Operation sequence or address mode sequence
Example: **ADD, AND, SUB**
Register Indirect
Indexed

Access Type
DR Data Read
DW Data Write
IR Instruction Read
NA No Access

ALU and CC (ALU function and Condition Code setting)
Example: ADD, AND, or OP condition codes
S Set
N Not set
X Don't care

Duplicates
(Not used in MIN)

Page and Location
(Not used in MIN)

Next State
BC Branch Conditionally
IB Instruction Branch
SB Sequence Branch
State ID Direct transfer

Access Width
(Not used in MIN)

Synonym
(Not used in MIN)

State ID
State Identification

■ indicates external bus activity

Figure 3.13 Format for a level 2 flowchart state

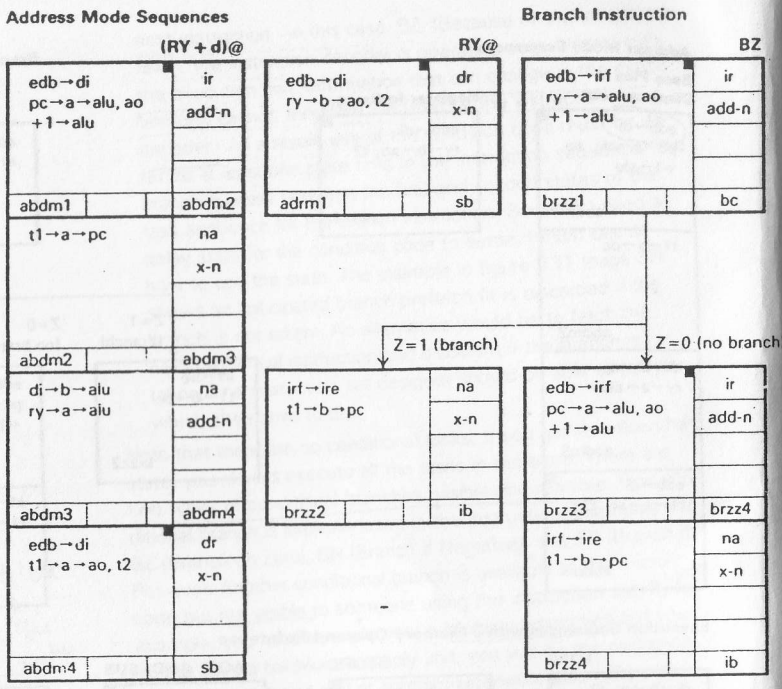


Figure 3.14 Format for final version of level 2 flowcharts (continues)

1. Sequence labels. These labels, associated with each execution sequence or address mode sequence, identify the instructions or address modes using that sequence. They also describe the transfer path. You will use this information later to build the instruction decoders. You can relate an instruction op code bit pattern to the first state ID in the address mode and execution sequence. For example, if IRE contains the bit pattern for register-to-register ADD, you want to begin instruction execution with state oprr1 (see page 45). In a microcoded controller, the instruction decoder will translate the bit pattern of the register-to-register ADD into the control store address of the control word oprr1. In a PLA decoder, the instruction bit patterns will form the AND array, and the control word state ID addresses in the control store will form

Execution Sequences with a Memory Operand Reference

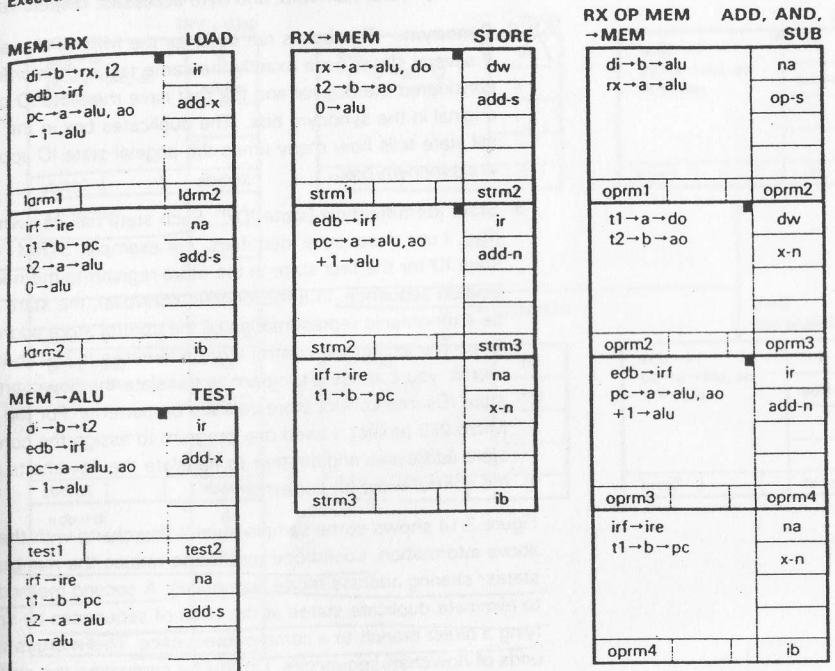


Figure 3.14 (continued)

the OR array. The PLA definition is derived from the labels in the flowcharts.

The label abbreviations are:

a	associated preceding quantity is an address
d	displacement
ADD	(for example) instruction using the sequence
MEM	MEMory
OP	OPeration
RX	source operand register
RY	address or operand register (see figure 3.3)

Execution Sequences for Register-to-Register and Special Instructions

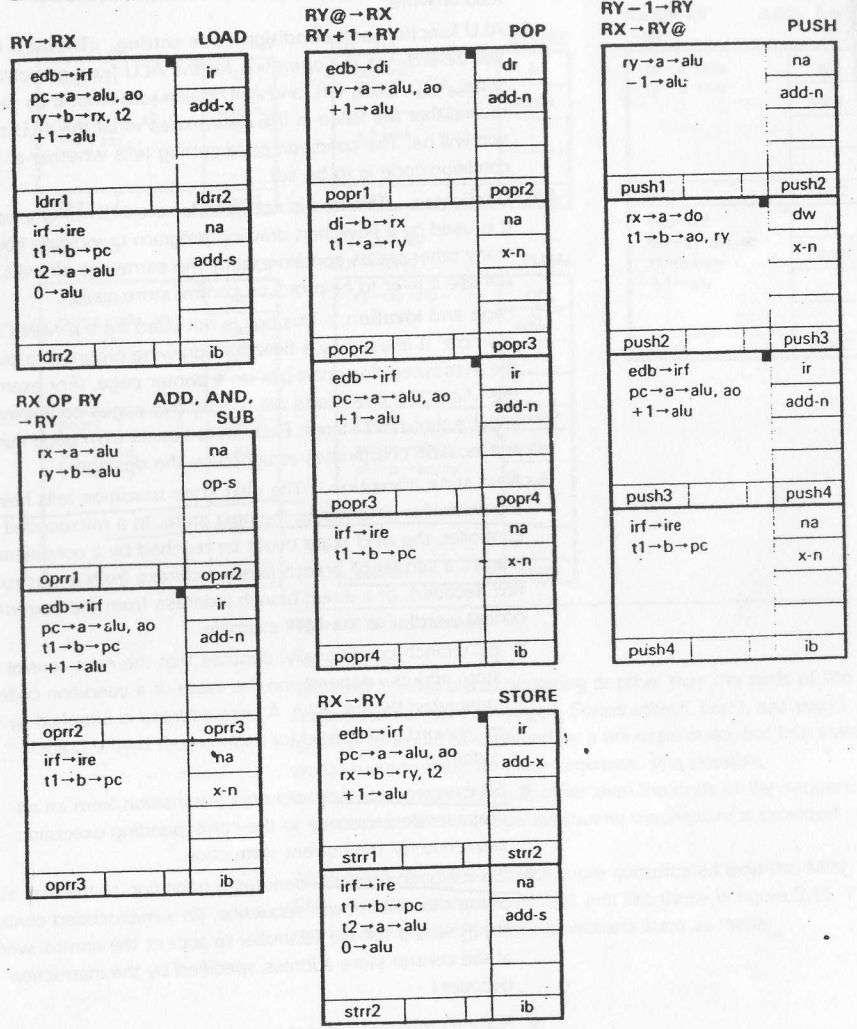


Figure 3.14 (continued)

2. **Access type.** This description says whether the controller is using the external bus for an instruction fetch or for a data read or write.
3. **ALU function and condition code setting.** The ALU function determines the operation for the ALU for a particular state. ADD, SUB, OR, and AND mean just what they say. OP means that the value in IRE determines what the ALU operation will be. The condition code setting tells whether a condition code is to be set.
4. **Duplicates.** This box is not used for the MIN CPU example. It is used by a flowchart drawing program to indicate how many other states contain exactly the same set of tasks. (We will use it later to help reduce control store size.)
5. **Page and location.** This box is not used for the MIN CPU example. It is used by a flowchart drawing program to place the associated flowchart box on a printer page. (For example, the Micro/370 flowcharts are twenty-five pages containing about a thousand states. Each state has its own page number and location coordinates assigned by the designer.)
6. **Next state transition.** The next state transition tells how the controller determines the next state. In a microcoded controller, the next state might be reached by a conditional branch, a sequence branch (a new address from an instruction decoder), or a direct branch (address from the current control word). For the MIN example:
   - BC (branch conditionally) denotes that the next control store address depends on the value of a condition code (generated by the ALU). A base address is supplied by the microword and altered (or augmented) by the branch condition.
   - SB (sequence branch) denotes a transition from an address mode sequence to the corresponding execution sequence for the current instruction.
   - IB (instruction branch) denotes a transition to the first state of the next instruction sequence. (In a microcoded controller, IB would tell the controller to access the control word at the control store address specified by the instruction decoder.)
   - State ID denotes a direct branch in the control store. The address of the next control word is in the current control word.
7. **Access width.** This box is not used for the MIN CPU example, but I would use it to indicate the size of the external bus

transaction. Micro/370 uses *w*, *h*, and *b* in this box to indicate external word, halfword, and byte accesses, respectively.

8. **Synonym.** This box is not used for the MIN CPU example. If several states have exactly the same tasks, then one is considered the original and the rest have the state ID of the original in the synonym box. (The duplicates box in the original state tells how many times the original state ID appeared in a synonym box.)
9. **State identification (state ID).** Each state has its own identifier. I use descriptive identifiers. For example, STRM1 is the state ID for the first state in the store-register-to-memory execution sequence. In a microcoded controller, the state ID will be a mnemonic representation of the control store address. Once you assign the control store addresses to the control words, you can use a program to translate the flowcharts' state IDs into control store address bit patterns. For the Micro/370 project, I used one program to assign the control store addresses and another to translate the flowcharts into the control word bit patterns.

Figure 3.14 shows some sample level 2 flowcharts with the above information. I used one method to reduce the number of states: sharing address mode sequences. A second method is to eliminate duplicate states at the ends of sequences by specifying a direct branch to a common sequence. This merges the ends of flowchart sequences. Do this by comparing the ending states of each sequence in figure 3.14 with all ending states below and to the right of the current sequence. Alphabetic organization of tasks, corner shadings, and access indicators make it *easier* to compare states. The result is figure 3.15 (pages 48-49), which has one-third fewer states than the flowcharts in figure 3.14. This is the most direct method of reducing controller size using flowcharts. When I did this for a CPU with hundreds of states (the MC68000), 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. Similar states have some tasks that differ. If you can make them the same without adversely affecting the associated sequences, you may eliminate some 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 the instruction parameters. (I did this in the ADD instruction example

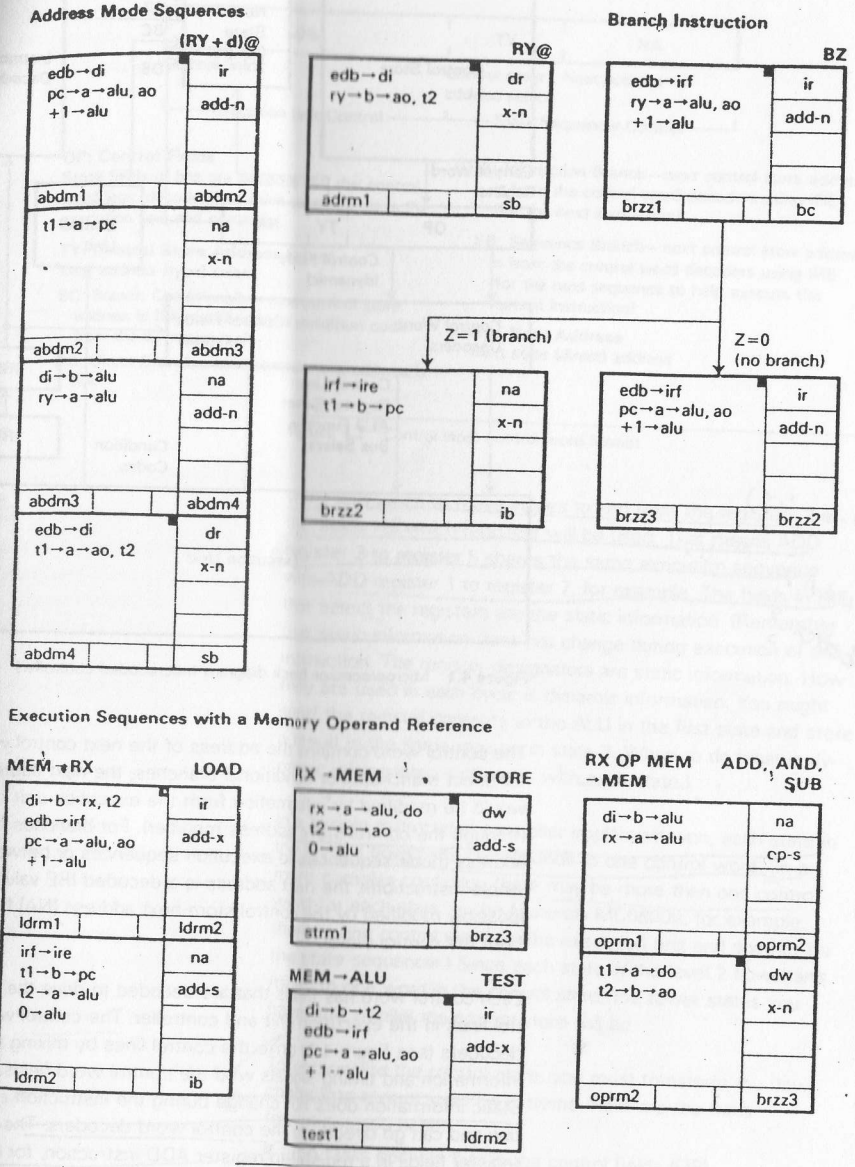


Figure 3.15 Merged level 2 flowchart examples (continues)

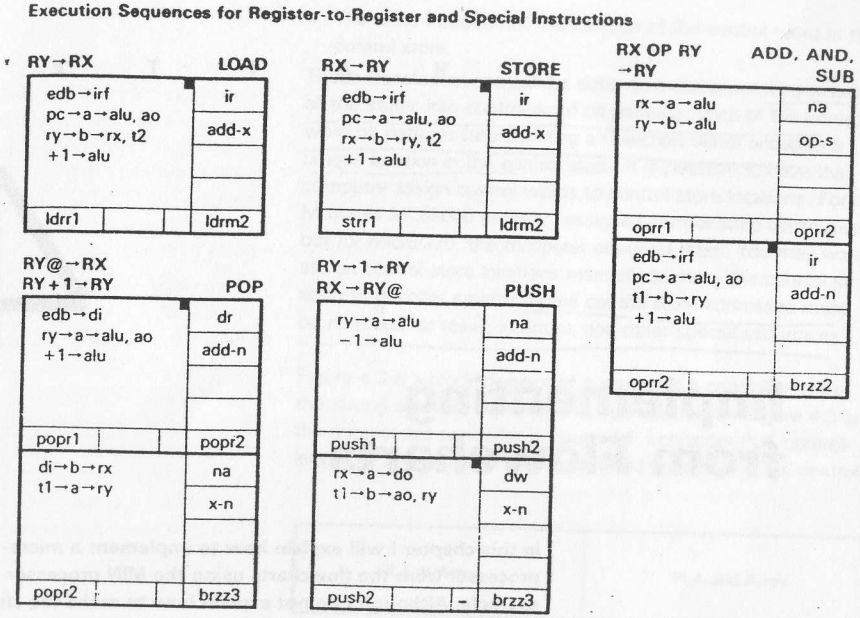


Figure 3.15 (continued)

for MIN.) Similar states occurring at other than the ends of the sequences cannot be merged. States adrm1, brzz1, and popr1 in figure 3.15 could be replaced by a universal state, but this state would have to exist for each sequence. It is possible to share common states at other than the ends of the sequences if some sort of microcode subroutine mechanism is provided.

Although a real CPU is much more complicated than the MIN CPU, the flowcharts would look just like those in figure 3.15. I implement my design from flowcharts such as these.

C H A P T E R

4

Implementing from Flowcharts

In this chapter I will explain how to implement a microprocessor from the flowcharts using the MIN processor example. Although I do not explain how to make the choice between (for example) a microcoded and a combinational design (see chapter 9 for the discussion of implementation methods), I will tell you how to implement the flowcharts for a microcoded, a PLA, and a combinational design.

Figure 4.1 is a block diagram of a simple microcoded controller, consistent with the function implied by the flowcharts. I show only enough detail for you to see the relationship between the controller and the flowcharts.

This is how the controller operates. An instruction is fetched (we'll worry about how some other time) and eventually placed in IRE. Translation of part of IRE's contents provides the control store address of the first word in the control word sequence for the instruction. Figure 4.2 shows the control store word format. Each flowchart state (see figure 3.15, pages 48–49) corresponds to a control word. The control word can specify register transfers (data and control registers), the ALU function and condition code setting, the source of the next control store address, and the next control store address.

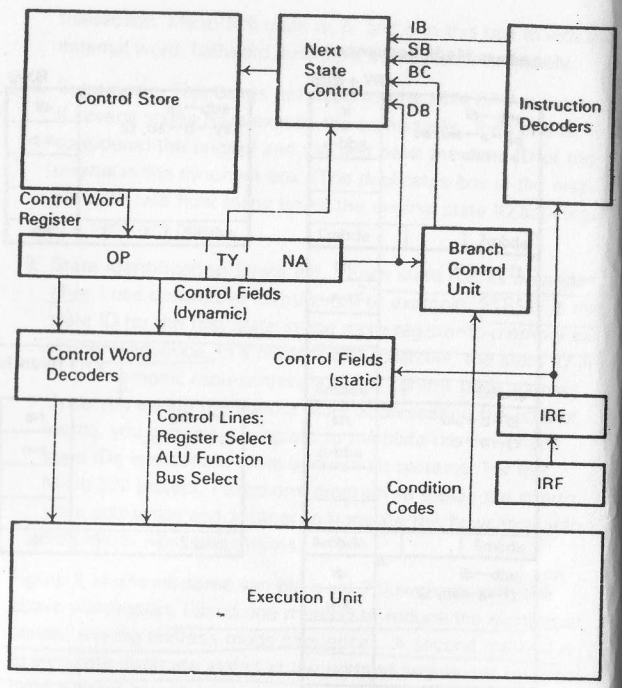
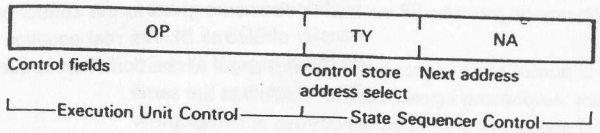


Figure 4.1 Microprocessor block diagram (microcoded controller)

The control word contains the address of the next control word for direct branches. For conditional branches, the next address would be modified by information from the execution unit (through the control store address modifier). 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 control store next address [NA] field from the control word).

Each control word has fields that are decoded to drive the control lines in the execution unit and controller. The control word decoders (see figure 4.1) drive the control lines by mixing static information and timing signals with the control word fields. Static information does not change during the instruction execution and can go directly to the control word decoders. The register fields in a register-to-register ADD instruction, for example, do not change during instruction execution. The control



OP: Control Fields
Small fields of bits are decoded (by the control word logic decoders) to drive control lines in the execution unit and controller.

TY: Control Store Address Select
Next address (type) select

BC: Branch Conditionally—next control store address is NA modified by a condition code from the execution unit

DB: Direct Branch—next control store address is NA

IB: Instruction Branch—next control store address is from the control word decoders using IRE (for the next instruction)

SB: Sequence Branch—next control store address is from the control word decoders using IRE (for the next sequence to help execute the current instruction)

NA: Next Address
Next state (direct) address

Figure 4.2 Control store control word format

word tells when to move values to and from the registers, but IRE fields tell which registers will be used. That means ADD register 3 to register 5 shares the same execution sequence with ADD register 1 to register 7, for example. The fields in IRE that select the registers are the static information. (Remember that static information does not change during execution of an instruction. The register designators are static information. How they are used in each cycle is dynamic information. You might send the register contents to the ALU in the first state and store a result in the same register in state 3. When to do what is dynamic information; it changes with each state.)

In a simple microcoded controller implementation, each state in the level 2 flowcharts corresponds to one control word. In a more complex controller, there may be more than one control word for each state. In the Motorola MC68000, for example, there is one control word for the execution unit and another for the state sequencer.) Since each state in the level 2 flowcharts maps to one word in the control store, the fewer states you have, the smaller the control store will be.

To personalize the control store, you must transform the flowcharts into control store bit patterns. Here are the transformations you need:

- The tasks become bits in the control fields (OP).
- The next state becomes the control store address select (TY) and next address (NA).

- The state ID becomes the location of the control word in the control store.

These transformations can be done on a computer. You translat all the states into control word bit patterns. Each of the control word bit patterns (representing a flowchart state) occupies a unique location in the control store. It is possible to have the computer assign control words to control store locations. For th Motorola MC68000 project, I assigned control store addresses, but for Micro/370, the computer assigned them. You may want assign control store locations manually to make the control store address decoder smaller. Some control store addresses must be reserved for reset, interrupt, and other special sequences.

Figure 4.3 is a block diagram of a simple PLA controller. Note the strong similarity between the PLA controller in figure 4.3 and the microcoded controller in figure 4.1. I consider PLA controllers to be a variation of the microcoded controller. If the control

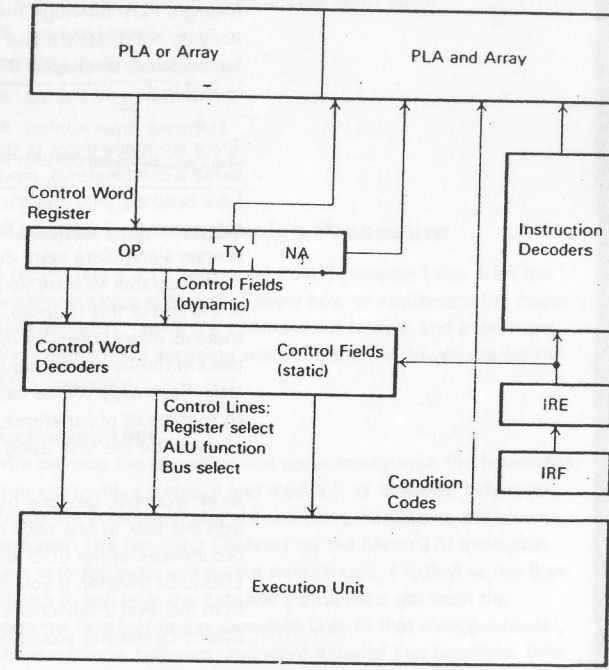


Figure 4.3 Microprocessor block diagram (PLA controller)

store address logic and the control store are implemented as an AND-OR PLA, the address logic would be the AND array and the control store would be the OR array.

Another way to see the similarity between a microcoded implementation and a PLA implementation is to consider the control store to be an orderly decode of an input address into a control word. If the control store address logic (address decoder, branch control unit, and multiplexer) of figure 4.1 produced the control word directly (instead of the address of the word in the control store), it would behave exactly like a PLA.

The flowcharts are used the same way except you may now be able to combine like states at other than 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 used for the microcoded controller. The PLA OR array contains the same bit patterns as the control store for the microcoded controller. Unused control store locations will be left out of the PLA. An apparent reduction in controller size may be possible using methods for splitting or folding a PLA. Although I will not discuss these methods here (they do not directly relate to using flowcharts), I do cover PLA folding briefly in chapter 8. I do not think it is a good idea to split or fold PLAs.

There are many ways to design a combinational controller (also called a combinatorial, random logic, or hardwired controller), but I will describe only one. First, design a state sequencer to duplicate the state transitions in the flowcharts. The flowcharts contain a complete state diagram. Techniques for converting state diagrams to state sequencers are known (see references at the end of this chapter), so I will not discuss them here. 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 DI in figure 3.15. Assign this task to one copy of the flowcharts by highlighting all occurrences of the EDB-to-DI transfer. Write the equation for the EDB-to-DI transfer. If you chose to implement the controller from the level 2 flowcharts of figure 3.15, the equation for the EDB-to-DI transfer would be:

EDIN = abdm1 + abdm4 + adrm1 + popr1

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

If you write the equations for unique states instead of individual tasks, you 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 I could group subexpressions of the equations to reduce the logic. The flexibility of the combinational design probably is the source of some of the trouble it causes.

Relationship between Flowcharts and Hardware

I view flowcharts as a compact, precise description of hardware requirements. Implementing a controller from a flowchart description is a logic design procedure. Here are the steps I recommend for implementing a microcoded controller from flowcharts:

1. **Execution unit.** The execution unit is developed concurrently with the flowcharts. You add things as you need them to develop the mature execution unit.
2. **Instruction decoders.** The instruction decoders translate an instruction bit pattern to the control store address for the execution sequence. For the MIN processor example, I need two of these decoders. The first decoder translates the instruction bit pattern into the control store address for the appropriate address mode sequence. The first decoder provides the address labeled IB in figure 4.1. If there is no address mode sequence, the IB instruction decoder points to the execution sequence. The second decoder translates the instruction bit pattern into the control store address for the execution sequence. The second decoder provides the address labeled SB in figure 4.1. Instructions that do not have an address mode sequence do not use the SB instruction decoder. Only the last state of each address mode sequence selects the address from the SB instruction as the next control store address. If you don't go to an address mode

sequence, you don't use the SB address, so you don't care where it points.

3. **Control word format.** The control word format is derived from the flowcharts. In most design procedures, the control word format is determined before the microcode is begun. I think that's wrong. How do you know what the control words need to do? If you do not have flowcharts, you guess. That's how you end up programming the function you need using the control functions you are given, rather than always having the control functions you need. If you have flowcharts, you can let them tell you the required capabilities of the control word precisely.
4. **Control word decoders.** Having the control word, you can then design the control word decoders. These decoders combine the control word (dynamic) control fields, the IRE (static) control fields, and timing signals, to provide the gate control signals for all transfers in the execution unit and the controller. For example, one field in the control word tells when a register value goes from the register to a bus; an IRE field tells which register is to be connected to the bus; and timing signals say when the transfer takes place. These signals are combined in the control word decoders to produce a single gate control signal (or signal pair).
5. **Controller block diagram.** A lot of people think that the block diagram should come first, but I disagree. When you use flowcharts, you are making assumptions about how to accomplish what you need to do; you are adding detail. The user's manual for an architecture tells at one level how an instruction is executed. It tells you that the Add Register instruction adds the contents of two registers and puts the result in one of the registers. The flowcharts tell you at the next level of detail how that instruction is executed. They tell how the register operands are connected to buses and the ALU, when they go to the ALU, and when the result returns.

When I designed the flowcharts, I made assumptions about register control, buses, and all the things needed to execute the instructions. Now, as I do logic design, I collect all the assumptions and implement the design. I get just what I need to do the job—no more and no less. If you pick a controller (as most textbook authors would have you do), how do you know what it needs to do? If you do not have flowcharts, you guess. If you do have flowcharts, the assumptions for the controller are in the flowchart description. Part of the logic design procedure is to extract the assumptions and build the controller.

Sample Design Chronology

Appendix B is a design chronology for logic design for the Micro/370 project. I began doing flowcharts in January 1981. I did the initial design of the input and output multiplexers in December 1981. I did the initial instruction decoders and the control word format in January 1982. I did not derive the control word format until the flowcharts essentially were complete. I did the initial controller block diagram in August 1982—about twenty months after I began working on the design!

Appendix B is taken from my notebook of weekly activities for the project, so I am not just trying to remember how I did it or construct how I think it should be done. I am telling you exactly how it has been done. I kept the same kind of notebook of weekly activities when working on the MC68000 design. The event chronology is essentially the same for that project. (Events were more compressed at Motorola because the MC68000 is simpler than Micro/370 and because Motorola emphasizes microprocessor design. Here are the steps again:

- execution unit
- instruction decoders
- control word format
- control word decoders
- controller block diagram

Sample Implementation Procedure

I repeat figure 3.15 here as figure 4.4 because I use it for the implementation examples. I show how to implement the instruction decoders, derive the control word format, and implement the control word decoders using the sample flowcharts for the MIN processor.

Execution Unit

You develop the execution unit concurrently with the flowcharts. You start with a proposal and modify it as required. When you finish the flowcharts, you also are done developing the execution unit. The last thing I defined for the Micro/370 execution unit was the input and output multiplexers. I looked at the flowcharts to see how the data and instructions got from the external data bus to the execution unit. In that microprocessor, there are byte, halfword, and word external bus transfers. Byte transfers can occur on any byte position on the external bus.

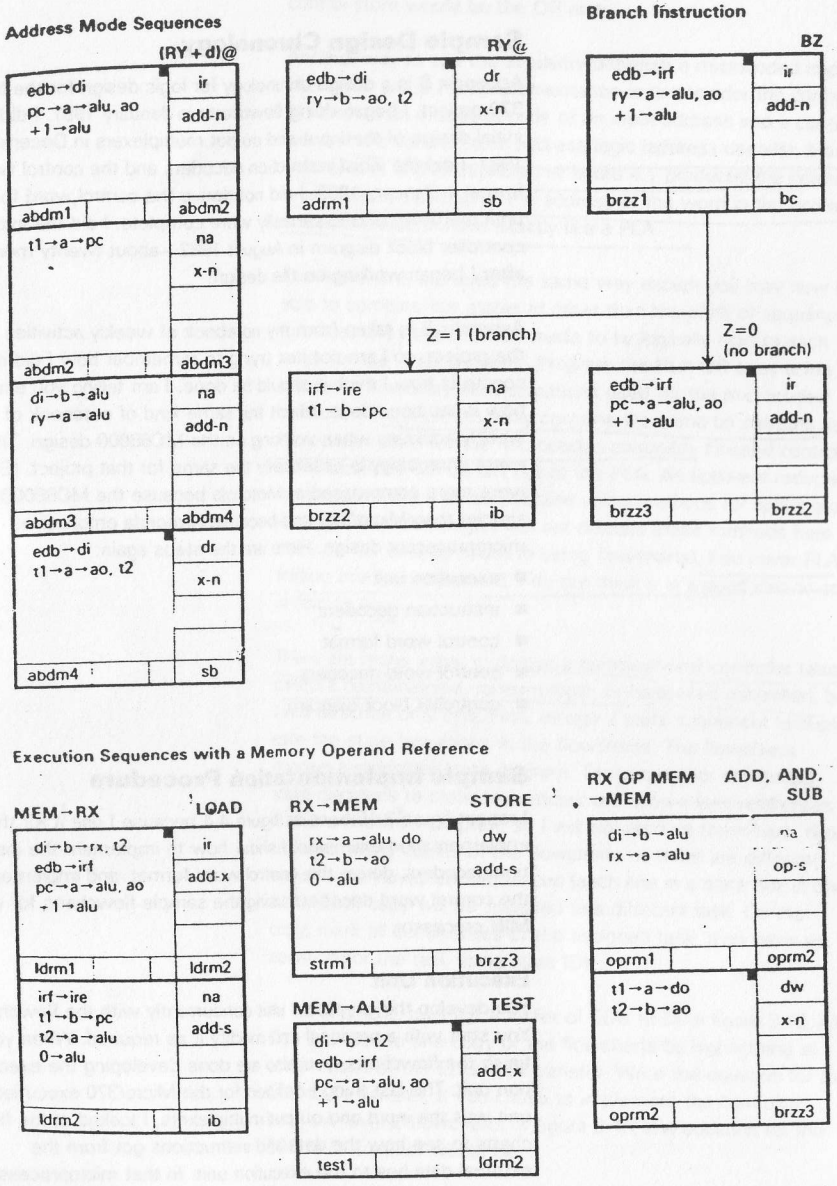


Figure 4.4 Merged level 2 flowchart examples (continues)

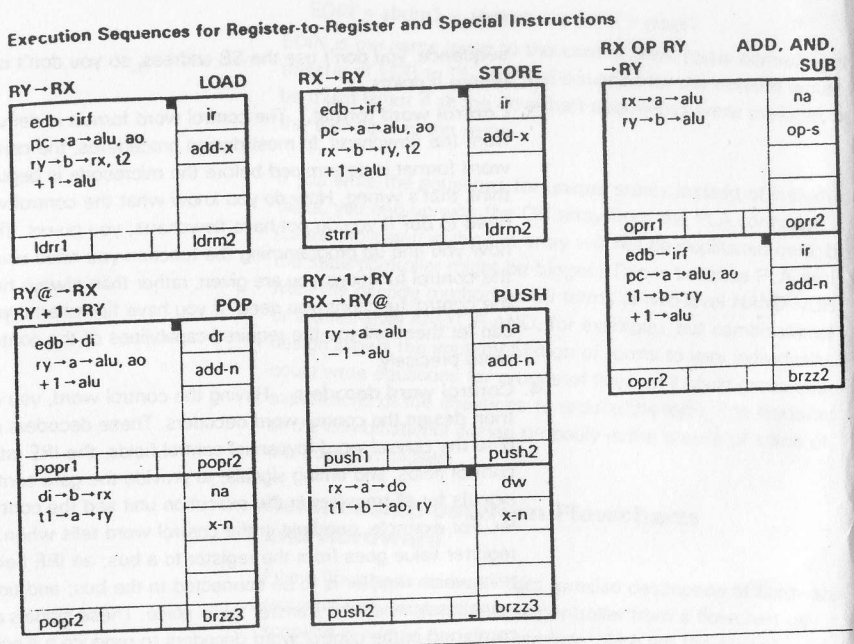


Figure 4.4 (continued)

The Micro/370 execution unit manipulates byte quantities only in the low-order byte position on the internal bus. I needed fancy multiplexers on the input and output connections to the external bus. The MIN processor does only word transfers, so it does not need the multiplexers.

Instruction Decoders

The MIN processor uses two instruction decoders. One decoder points to the first control word in an address mode sequence (if there is one), and the other points to the first control word in the execution unit sequence. The last state in any execution sequence shows IB in the next state block (see brzz2 and ldrm2 in figure 4.4). The IB instruction decoder points to the first control word in the next instruction sequence. The last state in any address mode sequence shows SB in the next state block (see abdm4 and adrm1 in figure 4.4). The SB instruction decoder points to the first control word in the execution sequence. For instructions with no address mode sequence, the IB instruction decoder points to the first control word in the execution se-

quence. (With no address mode sequence, the control word sequence and the execution sequence are the same. If there is no address mode sequence, it does not matter where SB points, since it is not used.)

Figure 4.5 shows sample instructions and the associated control word sequences, along with the IB and SB instruction decoder pointers. This shows how the flowcharts expect the instructions to execute. I designed instruction decoders IB and SB to map the path between instructions and from the address mode sequence to the execution sequence. One way to derive the mapping is to list all the instructions and then show the associated control word (if any) to which IB and SB should point. Another way to derive the mapping is to look at the first control word of each sequence and tabulate the instructions using that sequence (labels A and B) with the appropriate decoder. In practice, I use a third label on each control word sequence that tells

Instruction	Control Word Sequence	Next Control Word Address	IB Instruction Decoder	SB Instruction Decoder
POP	popr1	popr2	-	-
	popr2	brzz3	-	-
	brzz3	brzz2	-	-
	brzz2	-	abdm1	oprm1
ADD RX (RY + d)@	abdm1	abdm2	-	oprm1
	abdm2	abdm3	-	oprm1
	abdm3	abdm4	-	oprm1
	abdm4	-	-	oprm1
	oprm1	oprm2	-	-
	oprm2	brzz3	-	-
	brzz3	brzz2	-	-
	brzz2	-	oprr1	-
SUB RX RY	oprr1	oprr2	-	-
	oprr2	brzz2	-	-
	brzz2	-	adrm1	test1
TEST RY@	adrm1	-	adrm1	test1
	test1	ldrm2		-
	ldrm2	-	push1	-
PUSH				
.				
.				
.				

Figure 4.5 Example instruction execution sequences

me which instruction decoder points to the control word sequence. (Sometimes more than one decoder can point to the same sequence.) Figure 4.6 lists the control words to which the IB and SB decoders point for the flowcharts in figure 4.4.

The instruction mnemonics (such as AR, POP, L, and ST) correspond to bit patterns defined by the architects. The bit patterns should be listed in the user's manual. Even if you program the microprocessor in assembly language or FORTRAN using the mnemonics, the assembler or compiler translates the mnemonics into bit patterns. (You can store only ones and zeros in the memory, as these are the only values for which you can build logic.) The mnemonic control store addresses (such as oprr1, popr1, ldrm1, and strm1) also correspond to bit patterns (the real control store addresses). The instruction decoders translate the instruction bit patterns to the control store address bit patterns. If you build a PLA to implement the IB or SB instruction decoder, the bit patterns for the instruction mnemonics become the AND array, and the bit patterns for the corresponding control store address become the OR array (figure 4.6).

Control Word Format

I divide the control word into two sections based on function (see figure 4.2). The operation section, labeled OP, is composed of the fields for execution unit control. The next state section, containing TY and NA, contains the fields for state sequencer

IB Decoder Addresses	Instruction(s) or Address Mode	
abdm1	(RY + d)@	Address mode sequences
adrm1	RY@	
brzz1	BZ	Execution sequences (instructions **without** separate address mode sequences)
ldrr1	LR	
strr1	STR	
oprr1	AR, SR, NR	
popr1	POP	
push1	PUSH	
SB Decoder Addresses	**Instruction(s)**	
ldrm1	L	Execution sequences (instructions **with** separate address mode sequences)
strm1	ST	
oprm1	A, S, N	
test1	T	

Figure 4.6 IB and SB instruction decoders for MIN example

control. I divided the operation section into small fields that control the execution unit pieces. I did this by looking at the (logical) picture and simply assigning fields. To demonstrate the procedure, I have reproduced the MIN execution unit from figure 3.4 as a part of figure 4.7. Above that, I have allocated fields to the operation section of the MIN control word. There isn't much more to it than that. In chapter 8, I will show the much more complicated control words for Micro/370, but I used the same procedure to allocate the fields.

If two macros in the execution unit are never used at the same time, you might consider sharing the control field. This is done in a few cases for both Micro/370 and for the Motorola MC68000. Micro/370, for example, has one field that controls both the special Pack-Unpack unit and the shifter. The units do not have to have mutually exclusive use; you just have to be willing and able to do the extra control word decoding to separate the functions. Sharing control fields is not a good idea if the macros are not next to each other.

In a microprocessor, wire crossings are a big deal. You might save a field in the control word and waste more space crossing the signals to where you need them than you would have used by having an extra control field. In that case, you are trading area in the control store for area in the control word decoder. Carrying this to its extreme, let's say you save more and more fields in the control word. As you save fields, the control word decoder grows and the control word shrinks. When the control word is the same width as its input address, you don't need the control store. The result is a random logic implementation.

All I have done here is invent the format of the operation section of the control word. I have not assigned the number of bits to each field or made any bit pattern assignments. That's part of my logic design for the control word decoders: I look at the flowcharts to find out how many bits to assign to each field, and I use Karnaugh maps to help assign the best bit patterns to decode.

Control Word Decoders

Here is where I find out how many bits each control field needs and assign the bit patterns. I will show a few examples using the MIN processor flowcharts in figure 4.4. Here's the basic procedure:

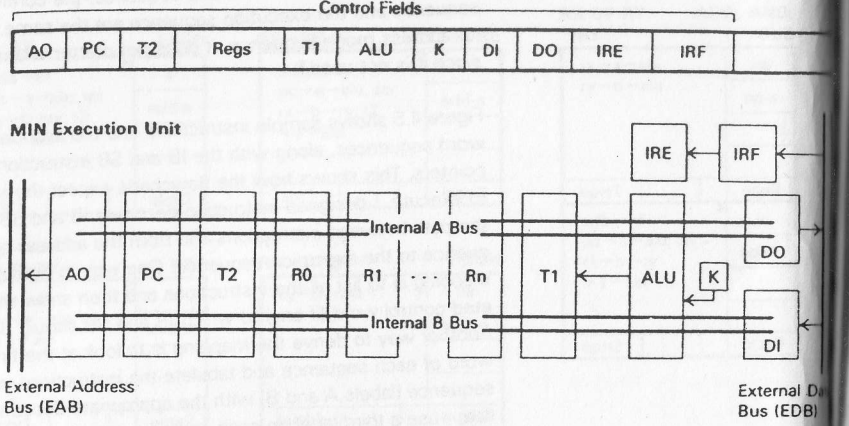


Figure 4.7 MIN control word (operation section) format

1. List uses of the macro.
2. Allocate bits.
3. Use a Karnaugh map to assign bit patterns.

That's it. Nothing difficult there. If you have your flowcharts in a computer data base, you can just call them up with the editor and collect all the occurrences of PC, T2, RX, or whatever. Once you have a list of all the different uses of a macro, you can assign the number of bits to the control field. If, for example, you did twelve different things with the Pack-Unpack unit, you would assign 4 bits to the control field. Don't forget to count "none" as one use. Some states do not use PC, for example, so you have to allow for those, too.

PC Control Example Look in figure 4.4 for all the ways PC is used. If PC goes to the bus, we don't care where it goes. If something writes to PC from the bus, we don't care where the data came from. We care only about controlling the paths into and out of the PC. To us, pc→a→alu looks the same as pc→a→rx. Here are the different uses of PC I have collected:

pc→a
a→pc (only one occurrence of this one)
b→pc
none

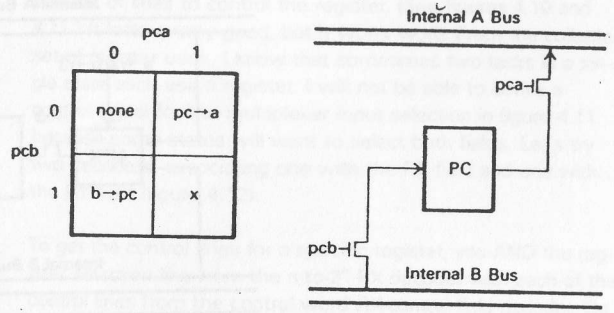


Figure 4.8 PC Karnaugh map and PC internal bus connection

Look again at the flowcharts in figure 4.4. There is only one occurrence of a→pc. I could just as easily write to PC using the B bus, so I can eliminate a→pc, thus simplifying the control logic. You probably will not be able to eliminate many of these uses in a processor with several hundred states, but I will do it here to simplify the example. With only three uses of the macro, I must assign 2 bits to the PC control field. Once I have the uses, I use a Karnaugh map to assign the bit patterns for the control field. Figure 4.8 is a diagram of the PC connection to the internal buses and the Karnaugh map for bit pattern assignment. One bit (pcb) will control an input connection to the PC from the internal B bus. The other bit (pca) will control an output connection to the internal A bus from the PC.

T2 Control Example I also looked in the flowchart examples of figure 4.4 for all occurrences of T2. Here's what I found:

t2→a
t2→b
a→t2 (only one occurrence of this one)
b→t2
none

This is about the same as the PC control example, as there is only one occurrence of a→t2. I can eliminate it, so I will. That leaves four uses (including "none"). I assign 2 bits to the control word field for t2. Figure 4.9 shows the internal bus connections for t2 and the Karnaugh map I used to assign the bit patterns. I can implement the logic for the control points using a 2-to-4 decoder. The 2 bits from the control word t2 field are the inputs. Three of the four output lines are connected (one each) to a control point; the fourth is "none" and is left unconnected.

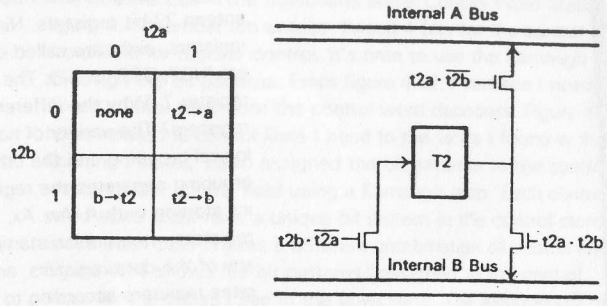


Figure 4.9 T2 Karnaugh map and T2 internal bus connection

Remember that the CPU logic works only with ones and zeros. When I want the CPU to do t2→b→alu, that's just what I put in the flowcharts. The CPU does not read the flowcharts, but I have to have somewhere to put the ones and zeros that eventually are transformed into the control signals on the gates.

The work I just did (in figure 4.9) tells me how to translate the flowchart statements into ones and zeros. The control store uses the ones and zeros of the T2 control field to drive the logic I just specified. That results in a signal on the appropriate control point (gate) in the execution unit. If I see t2→b→alu in the flowcharts, I should put 11 in the 2-bit T2 control field for that control word. If I do not use T2 in a particular state, then the T2 field for the corresponding control word should have 00 in it. What if a state contains two tasks using T2, such as t2→b→alu and rx→a→t2? Now what do I put in the T2 field? That case isn't listed among the ones I found in the flowcharts, so I didn't build hardware to do it. If you run into it, you have to change the state or change the hardware.

Register Control Example There's always a lot of confusion about register control, and it's easy to see why. The instruction has two ways to name registers. MIN calls one of the fields RX and the other RY (see figure 3.2). But the architecture contains only one set of registers: the programmer's register set. If there is only one set of registers in the architecture, there is only one set in the execution unit. RX and RY refer to the same set of registers.

In the MC68000 user's manual, for example, the writers refer to An, Dn, Rn, USP, SSP, SP, Ax, Ay, Dx, Dy, Rx, Ry, and Xi. I may

have missed one or two. Is it any wonder you get confused trying to design the control logic? There is only one set of seventeen 32-bit registers. Nine of the registers are called address registers, eight are called data registers, and two are stack pointers (USP, SSP, or SP). The stack pointers double as address register 7. Why the different names if there is only one set of registers? The variety of names is used in two ways. One is a logical grouping and the other is a physical grouping. The logical grouping separates the registers according to function. Mnemonics starting with A (An, Ax, and Ay) refer to any of the address registers, mnemonics starting with D (Dn, Dx, and Dy) refer to any of the data registers, and so on. The physical grouping separates registers according to which instruction field contains the register specification. The logical grouping tells you how to decode the instruction to tell whether you will be using an address or a data register. The physical grouping tells you where to find the field specifying the single referenced register in the logical group.

MIN has only one set of registers. RX means that you find the pointer to the referenced register in the first register field in the instruction. RY means that the register pointer is in the last field of the MIN instruction. It sounds as if all I need is a multiplexer to pick the register pointer from either RX or RY. Let's see. First, I collect all the flowchart references to RX or RY.

ry→a	rx→a; ry→b
b→rx	b→ry
ry→b; b→rx	b→rx; a→ry
rx→a	rx→a; b→ry
rx→b; b→ry	none

This collection of tasks looks (and is) more complicated than the set for T2. This time I had states with more than one task using the macro. Since all the tasks in the state are concurrent, I have to design the logic to be able to control two registers at a time. Next I look at the set of uses and draw how a typical register is connected to the internal buses. I did that in figure 4.10. I drew the connections to a typical register because I wanted to know the control points I would need.

R3 doesn't mean anything special; it's just one of the registers. Figure 4.10 tells me I need four lines from the control word decoders to control the registers. One line ($a \rightarrow r3/\overline{b \rightarrow r3}$) tells me whether the value written into the register comes from the internal A or the B bus. Another ($\rightarrow r3$) is active whenever a

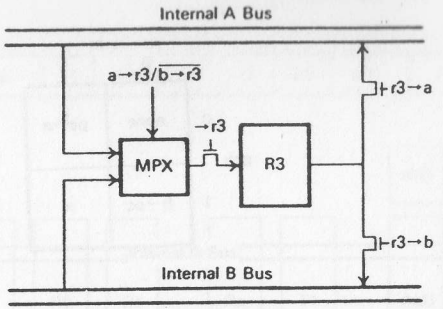


Figure 4.10 Connection of a typical register to internal buses

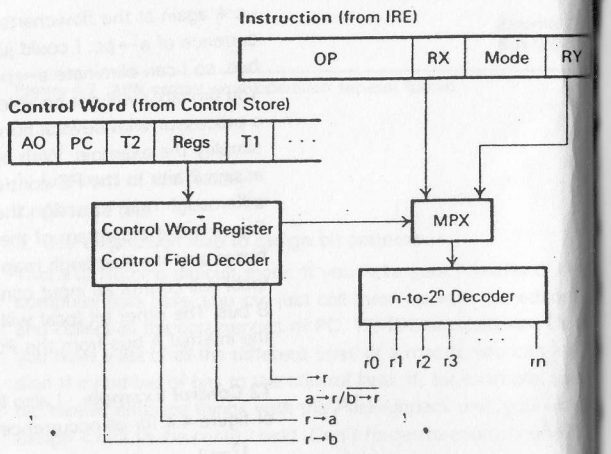


Figure 4.11 First proposed register control logic

value is written into the register. Two more (r3→a and r3→b) select the value in the register to one of the internal buses. The control word decoders should produce these four signals at the appropriate time. What about register selection? I intended to let the RX or RY field in the instruction select the register. The instruction is in IRE, so let's guess what the hardware might look like. Figure 4.11 was my first guess.

To get the control lines for a specific register, you AND the register selection line from the n-to-2^n decoder with each of the control lines from the control word register control field and use

that set of lines to control the register. (See figures 4.10 and 4.11.) It looks pretty good, but it won't work. From the collected set of register uses, I know that sometimes two tasks in a single state each use a register. I will not be able to derive a control signal for the multiplexer input selection in figure 4.11 because some states will want to select both fields. Let's try two decoders, associating one with the RX field and one with the RY field (figure 4.12).

To get the control lines for a specific register, you AND the register selection line from the n-to-2^n RX decoder with each of the control lines from the control word RX-control-field decoder. Do the same for the RY decoder and the control word RY-control-field decoder. Then OR these lines together. Here is what the control signals for register R3 look like:

$\rightarrow r3 = (b \rightarrow rx) \cdot x3 + (\rightarrow ry) \cdot y3$	load r3
$a \rightarrow r3 = (b \rightarrow rx) \cdot x3 + (a \rightarrow ry/\overline{b \rightarrow ry}) \cdot y3$	load from A
$r3 \rightarrow a = (rx \rightarrow a) \cdot x3 + (ry \rightarrow a) \cdot y3$	to A
$r3 \rightarrow b = (rx \rightarrow b) \cdot x3 + (ry \rightarrow b) \cdot y3$	to B

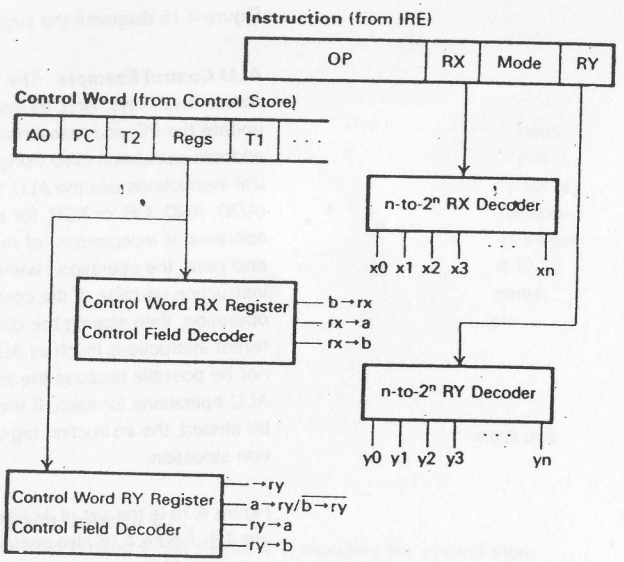


Figure 4.12 Second proposed register control logic

The occurrences in the flowcharts under Control Word States in figure 4.13 number ten entries. I need 4 bits for the control word field for register control. It's time to use the Karnaugh map to assign the bit patterns. From figure 4.12, I can see I need seven control lines from the control word decoders. Figure 4.13 helps relate the control lines I need to the tasks I found in the flowchart states. I also assigned the bit patterns to the control word register control field using a Karnaugh map. Each control word state must have a unique bit pattern in the control store (since each case implies a different combination of control lines). Figure 4.14 shows the bit patterns I assigned to the control store for the states I see in the flowcharts. The assignment of bit patterns to control fields is arbitrary. I try to assign patterns

Control Word States (What I see in the flowcharts)	Control Lines (Control signals from control word decoders)	Control Lines (Execution unit control points)
ry→a	ry→a	rx→a
b→rx	b→rx;→rx	ry→a
ry→b;b→rx	b→rx;→rx;ry→b	rx→b
rx→a	rx→a	ry→b
rx→b;b→ry	rx→b;→ry;b→ry	→rx
rx→a;ry→b	rx→a;ry→b	→ry
b→ry	b→ry;→ry	a→ry
b→rx;a→ry	b→rx;→rx;a→ry;→ry	b→rx
rx→a;b→ry	rx→a;→ry;b→ry	b→ry
none	none	

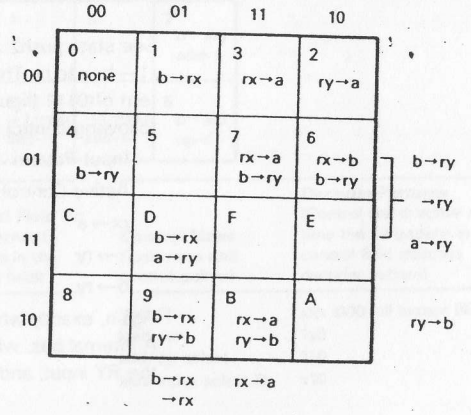


Figure 4.13 Relationship between the control word states and the control word register field decoders

Control Word States (What I see in the flowcharts)	Control Field Bit Assignments (The bit pattern I put in the control word register control field)
none	0000
b→rx	0001
ry→a	0010
rx→a	0011
b→ry	0100
rx→b;b→ry	0110
rx→a;b→ry	0111
ry→b;b→rx	1001
rx→a;ry→b	1011
b→rx;a→ry	1101

Control Lines	Decoder Patterns (Control line is active any time the control field matches decoder pattern)
rx→a	xx11
ry→a	0010
rx→b	0110
ry→b	10xx
→rx	xx01
→ry	x1xx
a→ry	11xx
b→rx	xx01
b→ry	01xx

Figure 4.14 Control word register control field bit assignments

that reduce control word decoders. Figure 4.14 also shows the equations for the control lines. The equations show how the control field bit pattern is decoded to control the execution unit resource (in this case, the register file).

Let's try a few examples to see whether the control does what I want. State abdm3 (figure 4.4) contains ry→a→alu. From figure 4.14, I see the register control field for that state should contain 0010. If I put 0010 in the control word, what control lines are active? Compare the control word bit pattern (0010) with each of the decoder patterns in figure 4.14. The only pattern that matches activates ry→a. Exactly what I wanted.

See state push2 in figure 4.4. It contains both rx→a→do and t1→b→ao,ry. That case corresponds to a control word bit pattern of 0111 (figure 4.14) for the register control field. The following control lines are activated by the 0111 bit pattern:

Input Pattern	0111
Active Control Lines	
rx→a	xx11
→ry	x1xx
b→ry	01xx

Again, exactly what I wanted. The RX register is enabled to the A internal bus, whatever is on the B internal bus is selected to the RY input, and I get a load signal to RY.

How about rx→b→ry,t2 from strr1 in figure 4.4? That corresponds to a control word register control field of 0110. (For any

state, I should find only one bit pattern that fits.) Here are the active control lines for a control word pattern of 0110:

Input Pattern	0110
Active Control Lines	
rx→b	0110
→ry	x1xx
b→ry	01xx

You should have the idea now. For each flowchart state, a unique bit pattern is assigned to the control word register control field. The control word decoder translates the bit pattern into signals that activate the required control lines in the execution unit.

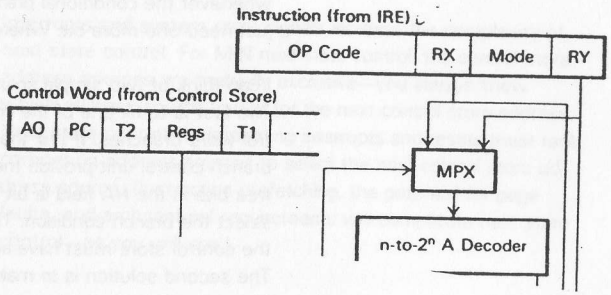
The register control I have shown works, but it is expensive. Control will be much cheaper if you:

- Assign one control word register-control-field decoder to each bus, rather than having a decoder associated with each register specification field, as in figure 4.12.
- Multiplex the input fields to the decoders for each bus.
- Do not allow transfers such as rx→*→ry (either bus).

Figure 4.15 diagrams the suggested register control logic.

ALU Control Example The flowcharts in figure 4.4 use the ALU in two ways. First, all instructions use the ALU for ADD to update the PC, and many instructions use the ALU for operand address calculation (also using ADD). Second, arithmetic and logical instructions use the ALU to perform the requisite operation (ADD, AND, OR, or XOR, for example). In the first case, the ALU operation is independent of the instruction context. In the second case, the operation I want from the ALU varies with the instruction op code. If the control word always specifies the ALU operation, then sharing the control word sequences among different instructions (such as ADD, AND, OR, SUB, and XOR) will not be possible because the control word must select different ALU operations for each. If the control word sequences are to be shared, the instruction register must participate in ALU function selection.

Figure 4.16 is the set of ALU controls from the flowcharts in figure 4.4. Figure 4.16 also contains a block diagram of the proposed ALU control logic and control points. The internal A bus is always an input to the ALU, and the output of the ALU goes to T1 for every ALU operation. The B side of the ALU gets



tants
it is
r set or
he flow-
l). I need
4.17
nents.
ple.

ntrol unit

Branch Control Unit

The branch control unit (figure 4.1) modifies the control store next address. State sequencing depends on conditions in the execution unit. This supports conditional branch instructions such as Branch on Zero (see BZ in figure 4.4). The microcode itself

Control Word States (What I see in the flowcharts)	Control Lines (Execution unit control points)
a→alu; +1→alu;add-n;alu→t1	load t1
a→alu; b→alu;add-n;alu→t1	load ccr
a→alu; 0→alu;add-s;alu→t1	add/op select
a→alu; b→alu; op-s;alu→t1	alub input select (2 bits)
a→alu; −1→alu;add-n;alu→t1	

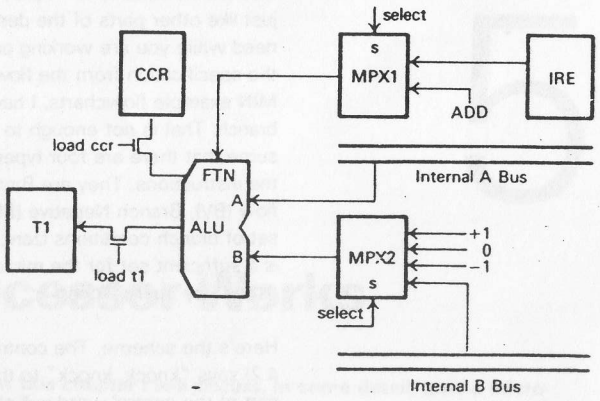


Figure 4.16 Proposed ALU control logic

Karnaugh Map for ALU Control Assignment

	00	01	11	10
0	0 none	1	3	2 b→alu add-n
1	4 0→alu add-s	5 +1→alu add-n	7 −1→alu add-n	6 b→alu op-s

Control Word States (What I see in the flowcharts)	Control Field Bit Assignments (What's in the control field)
a→alu; +1→alu;add-n;alu→t1	101
a→alu; b→alu;add-n;alu→t1	010
a→alu; 0→alu;add-s;alu→t1	100
a→alu; b→alu; op-s;alu→t1	110
a→alu; −1→alu;add-n;alu→t1	111
none	000

Control Lines (Execution unit control points)	Decoder Patterns (Control line is active any time the bit pattern in the control field matches decoder pattern)
load t1	xxx i000 (all except 000)
load ccr	1x0
add/op select	110
alub input select B	x10

Figure 4.17 Proposed ALU control word bit assignments

needs conditional next states for branches in the microcode. For example, if there is a multiply instruction and no hardware multiplier, the implementation of the multiply instruction uses a conditional branching algorithm. Instructions do not "see" these branches (except as different execution times for the instruction). Also, if the CPU checks for arithmetic overflow, you can achieve this with a conditional branch in the microcode.

Use the flowcharts to help design the branch control unit. It is just like other parts of the design: You assume whatever you need while you are working on the flowcharts, then you extract the specification from the flowcharts and build the unit. In the MIN example flowcharts, I have shown only one conditional branch. That is not enough to be a convincing example. I will assume that there are four types of conditional branches used in the instructions. They are Branch on Zero (BZ), Branch on Overflow (BV), Branch Negative (BN), and Branch on Carry (BC). This set of branch conditions (zero, overflow, negative, and carry) also is a sufficient set for the microcode (whose branches are not "seen" by instructions).

Here's the scheme. The control word (TY field, shown in figure 4.2) says "knock, knock" to the branch control unit. Some other part of the control word will choose the branch condition of interest. This part of the control word says to the branch control unit, "Is *this* branch condition (e.g., overflow) there?" The branch control unit just says yes or no. Since there are four possible branch conditions (using the Z, V, N, and C condition code bits from the ALU), I need two bits to choose the branch condition. I do not need a "none" in this case because the control word (TY field) will not select the branch control unit's output unless there is a conditional branch. Because each of these branch conditions is indicated by a single-bit value, I need only a 1-bit output from the branch control unit. A simple way to implement this branch unit is to use the ALU condition bits (Z, V, N, and C) as inputs to a 4-to-1 multiplexer and to let the control word select the branch condition. The branch control unit output bit (BC) is substituted for the low-order bit from the NA field.

Where in the control word is the field that selects the branch condition? I could add one (2-bit) field to the control word format. But that would add 2 bits to every control word, even though branch-type control words are not used very often. We don't want to do that. Since one control store address bit comes from the branch control unit, there is an extra bit in the NA field

whenever the conditional branch is specified (by the TY field). I just need one more bit. Where can I get it?

I can think of two simple solutions, both involving bit-robbing. The first is to fix one of the control store address bits for all control word branches. If I fix the high-order bit at zero and let the branch control unit provide the low-order bit, then there are 2 free bits in the NA field (a bit at each end) that can be used to select the branch condition. This means that all branch targets in the control store must have an address with a high-order zero. The second solution is to make the branch control unit have a 2 bit output. If the branch control unit provides 2 bits, there are 2 free bits in the NA field again. I don't even have to make the branch control unit any more complex. I could use its single output line for the 2 low-order address bits for the next control store address. Both methods restrict addresses for control words that are branch targets. That is, they complicate control word "placement" in the control store.

Next State Control

Next state control for MIN is a simple multiplexer. I have four sources for the next control store address: two from the instruction decoders, one from the branch control unit, and one from the control word. The control word (TY field) selects the next control store address (figure 4.18). The next state control for MIN is simpler than the next state control for a commercial microprocessor. Next state control for MIN implements four next state sources, while Micro/370, for example, implements nine. MIN does not implement interrupts or system reset signals.

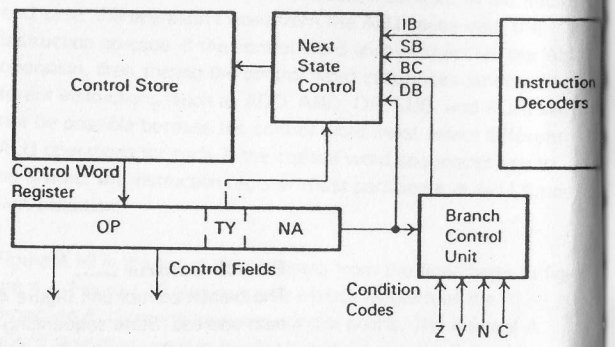


Figure 4.18 Control store branch control and next address select

Interrupts and system reset signals increase the complexity of next state control. For MIN next state control, the control store address sources are mutually exclusive—you always know which address source to use for the next control store address. Next state control implementing interrupts and resets must rank competing address sources to select the next control store address source. (Instruction prefetching, the potential for page faults, and architectural requirements will complicate next state control—as you will see.)

Summary

The format of all the control word fields is determined as above. Start at the left side of the execution unit and define the control word fields for each macro until you get to the right side. Use the flowcharts to tell you what you need. The execution unit tells you the number of fields, while the flowcharts determine how many bits to assign to each field. Draw a block diagram of the macro and the control for the macro to determine number of control lines. Then use a Karnaugh map to assign control field bit patterns in a way that minimizes the control word decoders.

The flowchart method transforms the English-language specification of the CPU into a formal description of how the CPU behaves. Use the flowcharts to construct only the hardware you need to implement the specified CPU. Derive the control word format from the flowcharts. You need not guess what the requisite capabilities of the controller must be or what the desired format and capabilities of the control word are.

References

Christopher R. Clare, *Designing Logic Systems Using State Machines* (New York: McGraw-Hill, 1973).

William I. Fletcher, *An Engineering Approach to Digital Design* (Englewood Cliffs, New Jersey: Prentice-Hall, 1980).

Zvi Kohavi, *Switching and Finite Automata Theory* (New York: McGraw-Hill, 1970).

John B. Peatman, *The Design of Digital Systems* (New York: McGraw-Hill, 1972).

Charles H. Roth, Jr., *Fundamentals of Logic Design*, 2d ed. (St. Paul, Minnesota: West Publishing Co., 1979).

David Winkel and Franklin Prosser, *The Art of Digital Design* (Englewood Cliffs, New Jersey: Prentice-Hall, 1980).

C H A P T E R

5

How a Microprocessor Works

In this chapter I will discuss, in some detail, how a microprocessor works. This procedure will be like taking an auto engine apart and examining it piece by piece. You will get to see a real crankshaft—and believe me, it doesn't matter that it's from a 289 Ford. Seeing the real thing gives you a reference point for asking questions about why things are done a certain way. I will talk about problems in designing logic for the IBM System/370 architecture. It would be too bad if your reaction is "that's for the stupid 370 . . ." because I think the System/370 is, in fact, an outstanding architecture. Study the solutions. I will slip in explanations of IBM Micro/370 and Motorola MC68000 parts to prepare you for chapters 6 and 7.

Figure 5.1 is a diagram of a microprocessor. Assume that the microprocessor is running a program. I view instructio execution as a composition of three parts: fetch, decode, and execute. The simplest CPU would work serially: fetch the instruction, decode it, execute it, then fetch the next i struction. In more complicated CPUs, fetch, decode, and execute happen at the same time. While the CPU is execu ing the current instruction, it is decoding the next and fetching the one after that. Let's see what is happening

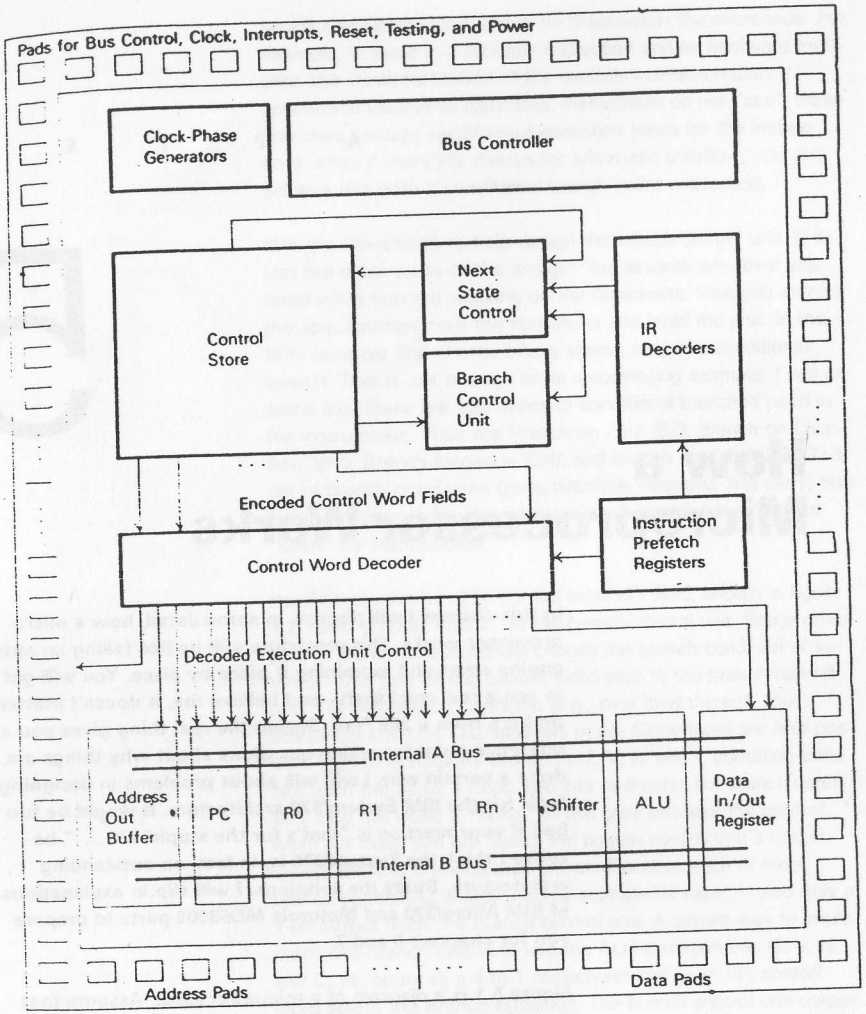


Figure 5.1 Microprocessor chip

while a program is running. Assume that the microprocessor has just finished decoding the current instruction.

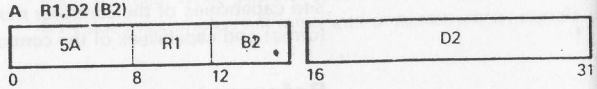
The instruction format is shown in figure 5.2 (which appeared earlier as figure 2.5). 5A (hex) is the op code for ADD. The R1 field designates a general register where the first operand is found and where the result will be stored. The B2 field designates the general register to be used as a base address register. The D2 field designates a displacement value (halfword) to be added to the contents of the base register to form the address of the second operand.

Here's what has to happen to get the microprocessor to the same point in the next instruction.

1. Fetch the remaining instruction halfword.
   The incremented PC value is placed on the address pads, and the controller waits for the halfword. The returning halfword is saved in a temporary register.
2. Calculate the operand address.
   The register designated in the B2 field is added to the contents of the D2 field.
3. Fetch the operand.
   The calculated operand address is sent to the address pads, and the state sequencer waits for the operand halfword to return. It is saved in a temporary register.

ADD

A R1,D2 (B2)



The second operand is added to the first operand, and the sum is placed in the first-operand location. The operands and the sum are treated as 16-bit signed binary integers. The first operand is in the register specified by the R1 field. The second operand is in memory. The address of the second operand is formed by adding the displacement specified by the D2 field to the contents of the base register specified by the B2 field.
An overflow causes a program interruption when the fixed-point overflow mask bit is 1.

Resulting Condition Code	Program Exceptions
0 Sum is zero	Access (fetch)
1 Sum is less than zero	Fixed-point overflow
2 Sum is greater than zero	
3 Overflow	

Figure 5.2 The ADD instruction

4. Add.
   Add the contents of the register designated by the R1 field to the contents of the temporary register. The answer is held in the ALU.
5. Store the answer.
   The result is sent to the register designated by the R1 field.
6. Update the PC.
   The PC is incremented to point to the next instruction halfword. It is saved during the next state.
7. Fetch the first halfword of the next instruction.
   The new PC value is sent to the address pads, and the state sequencer waits for the instruction halfword.
8. Find the control store address of the next instruction's control word.
   The next instruction halfword is sent to a register at the input to the instruction decoders, which are called IR (instruction register) decoders.
9. Branch to the next instruction control word sequence.
   The IR decoders produce the address of the next instruction's control word sequence. The last state in the control word sequence for the ADD instruction changes the next state control (figure 5.1) to choose the next control store address from the output of the IR decoders.

Each of the states in the sequence is represented by a control word in the control store. The control word format is shown in figure 5.3. Each control word contains encoded control patterns for control of the execution unit, bits to select the next control store address, and a field for the next control store address. In the control word sequence for the ADD instruction, each of the control words except the last contains the address of the next control word. The last control word selects the address at the output of the IR decoders.

In a commercial microprocessor, many of these steps overlap—that is, occur at the same time. It takes one cycle to decode the instruction (make the control store address); only the IR decoders are used. During that time, you can use the execution unit and the external bus. I use the external bus to read or write data

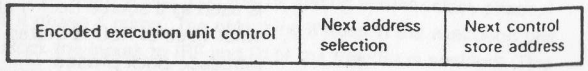


Figure 5.3 Control word format

State	Execution Unit	Decoder	External Bus
1			Read instruction halfword
2	ALU = D2 + (B2)		
3			Read operand halfword
4	ALU = (DI) + (R1)		
5	R1 = (ALU)		
6	ALU = (PC) + 2		
7	PC = (ALU)		
8			Read instruction halfword
9		IR	

Figure 5.4 ADD instruction execution with no overlap

State	Execution Unit	Decoder	External Bus
1	ALU = D2 + (B2)		Read instruction halfword
2			Read operand halfword
3	ALU = (PC) + 2		
4	PC = (ALU)		
5	ALU = (DI) + R1)		
6	R1 = (ALU)	IR	Read instruction halfword

Figure 5.5 ADD instruction execution with overlap

or prefetch another instruction halfword. I use the execution unit to put a result in a register or to update the program counter. Overlapping operations speeds instruction execution (I use processor resources concurrently rather than serially).

Figure 5.4 shows the resources of the microprocessor and the states of the ADD instruction, without overlap. See how sparsely the resources are used. The idea of overlapping is to use the independent resources concurrently to complete the job in fewer states.

Figure 5.5 shows how the ADD instruction can be overlapped. I made subtle changes in the way the microprocessor executes instructions. The calculations are all the same, and I still have the same number of external bus accesses, but the assumptions are different. The instruction halfword being fetched in state 1 cannot be the second halfword of the ADD instruction, because I assumed that halfword was in the microprocessor when I used the displacement in D2. The halfword fetched in state 1 is the first halfword of the next instruction. The halfword fetched in state 6 is either the second halfword of the next instruction or the first halfword of the instruction following the next instruc-

tion. This works okay for a series of ADD instructions (as in figure 5.5) because I need to have the second halfword in the microprocessor at state 1 and the sequence for the ADD puts it there.

This design looks better, but it still is not complete. There are two instruction accesses in the sequence, but the instruction address (PC) is only incremented once. Further, there is no place to save the extra instruction halfword between instructions. These must be added. There isn't any way to accomplish the PC increment without increasing the number of states in the sequence—unless I add some hardware for that, too. I have already said that a commercial microprocessor could do the instruction in five states, so let's keep working on this till we get there.

First, I need a more accurate representation of the execution unit. I'm going to change notation, too. It's getting too difficult to keep track of everything that is going on in a single cycle. I will use a box to show one state. Inside the box, I will list the tasks that occur during that state. These tasks occur at the same time, so I list them in alphabetical order for convenience. In our microcoded controller, each state will be represented by one control word. A sequence of states will implement an instruction. Figure 5.6 is an execution unit with enough detail to see what the processor is doing to execute the instruction. Figure 5.7 is the ADD sequence of figure 5.5 using the new notation.

This control word sequence appears to do everything in five states. I added hardware to improve performance. Now there is

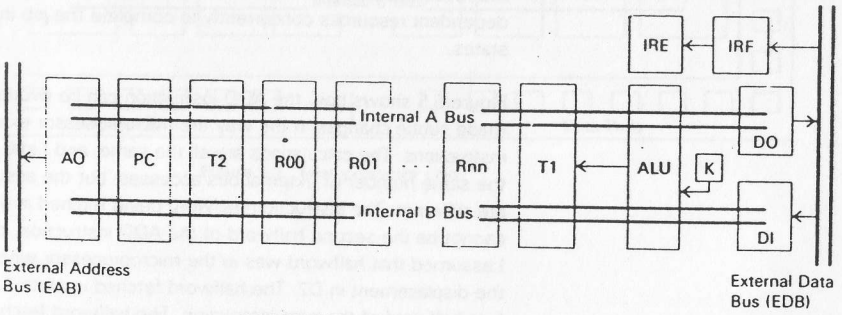


Figure 5.6 A more detailed execution unit

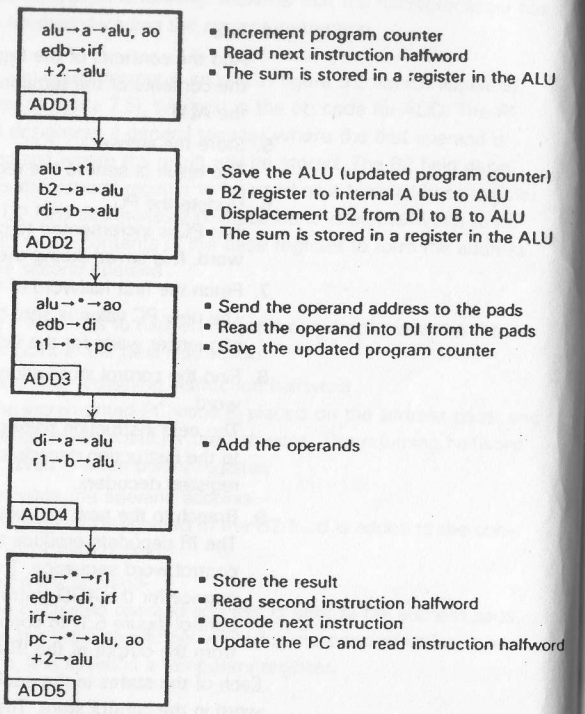


Figure 5.7 ADD instruction execution with overlap (state notation)

a new instruction register for fetch (IRF) to hold the next instruction while the current instruction is being executed. The ALU is directly connected to a temporary register to allow simultaneous calculation and saving of the previous result. Am I done? No. The problems are subtler.

Look at state ADD5, where I am fetching the halfword following the next instruction, decoding the next instruction in the IRE, incrementing the program counter, and saving the result of the ADD. If I change the value of IRE, the answer for the ADD instruction might get put in the wrong place because the IRE holds the pointer to R1. State ADD2 also looks funny. The ALU is used as a source (to T1) and as a destination. How do I know what value of ALU is stored in T1? To solve these problems, I introduce clock phases.

Internal Clocking

I said that all the tasks in a state happened at the same time. I lied. They can't. It depends on how closely you look. From an external view, everything in the state appears to happen at the same time. (If you are outside the chip, you can observe only what is happening at the pads.) Inside the chip, you have to have a clocking scheme to regulate events. One common clocking scheme uses a four-phase clock.

During phase 1, the source register is gated to the internal bus. During phase 2, the signal on the internal bus is amplified and broadcast the length of the bus. During phase 3, the signal on the internal bus is gated to the destination. In phase 4, the bus is returned to a neutral state. In state add3, for example, the register T1 is sent to one of the internal buses in phase 1. In phase 2, the contents of the register are amplified and broadcast down the bus. During phase 3, the gate to the PC register is opened and the signal on the bus overwrites the contents of the PC.

In state ADD4, the register DI and the register designated by the R1 field of instruction are gated to buses B and A, respectively, during phase 1. During phase 2, the contents of the registers are amplified and broadcast down the bus. During phase 3, the inputs to the ALU are opened, and the ALU begins to operate. During phase 4, the result of the ALU operation is saved in the ALU output register, and the resulting condition codes are sent to the condition code register.

That's the way it's supposed to work. Actually, however, the ALU is part of the critical path in the execution unit, so it is always connected to the internal buses and always operating, but the result is saved (at phase 4) only for the states that use the ALU. It gives the ALU a head start and explains why the register associated with the ALU is at the ALU's output and not at the ALU's input.

That's the clocking scheme—almost. There is one more piece—memory. I put addresses out, but I don't know when the data will come back. The clocking on the chip has to allow for delays from the external bus. I do this by adding another clock phase. I call it phase 4 prime. The only thing it does in the execution unit is clock the inputs to IRF and DI (if the transfer is enabled). This lets me compute the result once and wait for the memory data

for as many clocks as is necessary—without affecting other parts of the execution unit state

Let's look at the internal timing of the figure 5.7 states to see whether everything is working okay. In state ADD1, the PC will reach AO at phase 3. AO is the buffer that drives the pads. The instruction returned by the memory on the EDB will arrive at phase 4 or (some) phase 4 prime. In ADD2, the value being saved in T1 is the updated PC. The ALU-to-T1 transfer must be a phase 1 transfer. The transfers in ADD5 are a problem. The next instruction is in IRF and must be moved to IRE before the next halfword is loaded at phase 4 (or phase 4 prime). But the register pointer (R1 field of the instruction), used in phase 3, depends on the contents of IRE. The problem in ADD5 is still there. I am changing IRE in phase 1, but I need the R1 field in phase 3. I will postpone fixing this problem until I discuss controller timing.

Timing between the Execution Unit and the External Bus

Now that it looks as if I have the timing down pretty well, here is a rule of thumb I use for today's technology. I can execute two internal states (microcycles) for every external bus access. (Two CPU cycles per bus cycle.) This means I can put the address on the pads in one state and look for the returning data at the end of the next state. I call these split cycles. The people who buy your chip can measure performance only at the pads. The more processing you hide inside the memory references, the faster the chip looks. Two microcycles per external bus access is a rule of thumb I have used since 1977; the rule is still valid.

Instruction Execution Time

To assess the execution time of the instruction, I count the state as one unit if it has no external reference and two units if it has an external reference. I call these units microcycles. The instruction sequence in figure 5.7 takes eight microcycles to complete (even though it uses only five control words). Split cycles count as one unit each because they distribute the memory delay over two internal states. Let's see how I can improve the performance of the processor by taking advantage of split cycles. Figure 5.8 is the same ADD instruction but uses split cycles to improve performance.

The ADD instruction in figure 5.8 takes seven microcycles. I split only the first instruction access. I could have split the other two, but it would cost two control words and the performance would not improve. I am doing well. There is only one state in the figure 5.8 sequence that is not hidden from the outside observer by an external bus cycle. Since the ADD instruction is important, I cannot let this happen. I will add hardware to make the instruction faster. I could save another microcycle if there was a way to get the next PC to the pads while the ALU is doing the ADD. Let's just add the hardware to do it.

The execution unit of figure 5.9 is the same as figure 5.6 except the internal A bus is now segmented into two pieces (called AP and AD). I can dynamically control the connection from each

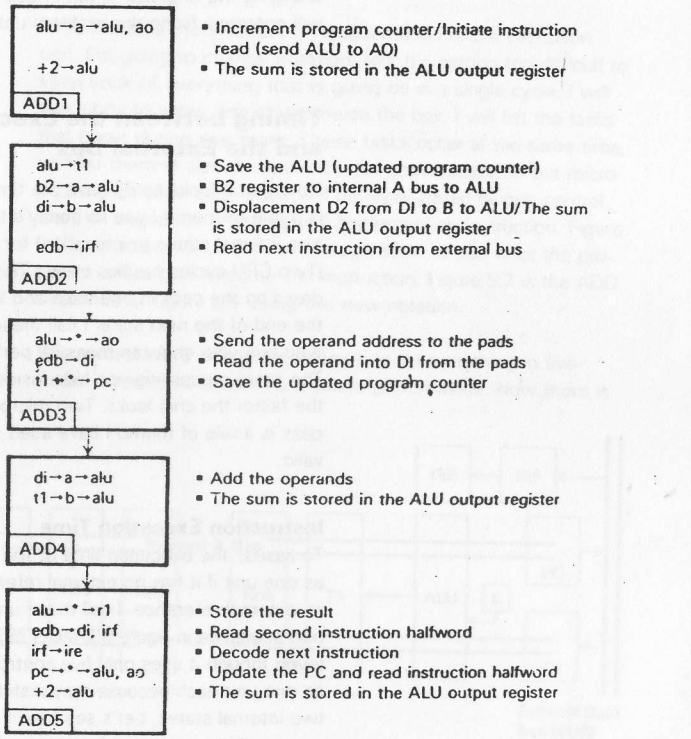


Figure 5.8 ADD instruction execution using split cycles

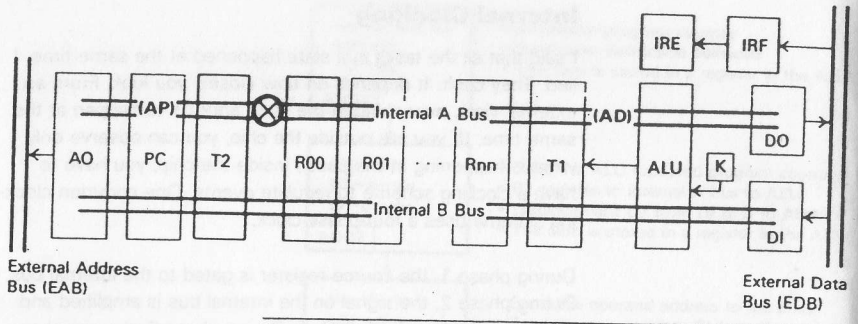


Figure 5.9 A more detailed execution unit with a bus coupler

control store word. Now I can eliminate one more microcycle from the ADD control word sequence (figure 5.10). I can't do any better as long as I am limited to a single external bus for instructions and data. This is almost like the hardware for a real commercial microprocessor execution unit. Mine is simpler because I want to show only how it operates. In practice, the execution unit will be more complicated because special hardware accommodates a greater variety of instructions, but the principles of timing and execution stay the same.

Global Chip Layout and Timing

Now it's time to back away from the execution unit a little to look at timing for the whole chip. If I look at what is occurring in the execution unit and on the external bus, I can deduce the timing for the rest of the chip. Figure 5.11 is a block diagram of a microprocessor. It shows the major communication paths for instructions, addresses, and data. Notice how the information comes in at the lower right corner and flows around the chip in a counterclockwise loop. All external addresses and data pass through the pads around the edge of the execution unit. Data into and out of the execution unit pass directly to and from the pads nearest the lower right corner of the chip. Instructions enter the execution unit on the data pads and are passed either to a temporary register in the execution unit (in the case of a displacement) or to the instruction decoder (in the case of an op code).

The output of the instruction decoders is the control store starting address for the control word sequence corresponding to the

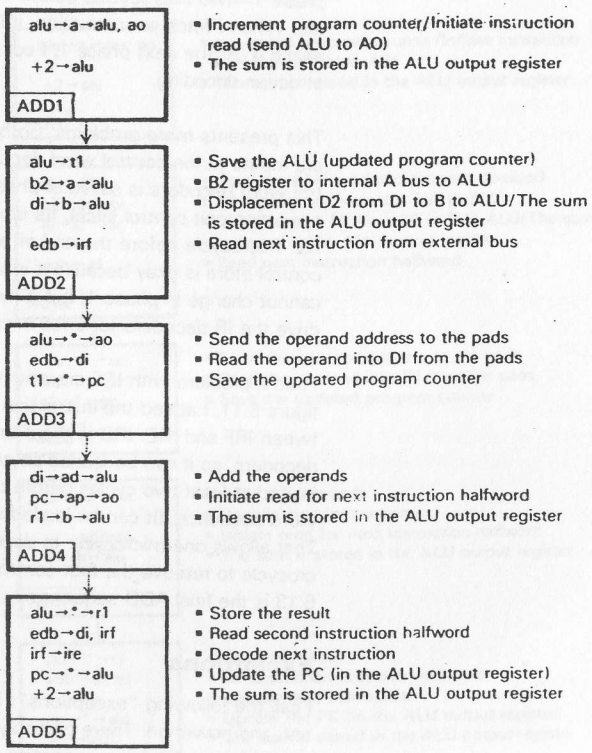


Figure 5.10 ADD sequence with segmented internal bus

input op code. The address goes to the next state control and then to the control store. Part of the control word leaves the control store at the top, and part of it leaves at the bottom. At the bottom are the bits controlling the execution unit, and at the top are the bits for next state control. The control store bits leaving the bottom of the control store are decoded to control the execution unit.

The control store is used only for signals that change every state, but some of the information to do an instruction stays the same during the execution of the instruction. If the register pointers, for example, do not change during the instruction, there is no need to translate the register fields through the control store. The register pointers can be taken directly from the

IRE. The IRE is above the execution unit and directly feeds the control word decoders. Nothing on the chip is physically located very far from where it is used. The overall information flow on the chip is a loop. This loop is characteristic of the layout planning for a sensible design. It reduces wiring and delays. Since wiring tends to follow the flow of information, a loop means fewer wire crossings, too.

I already know some of the timing for the external buses and the execution unit, so let's figure out the rest of it. First I have

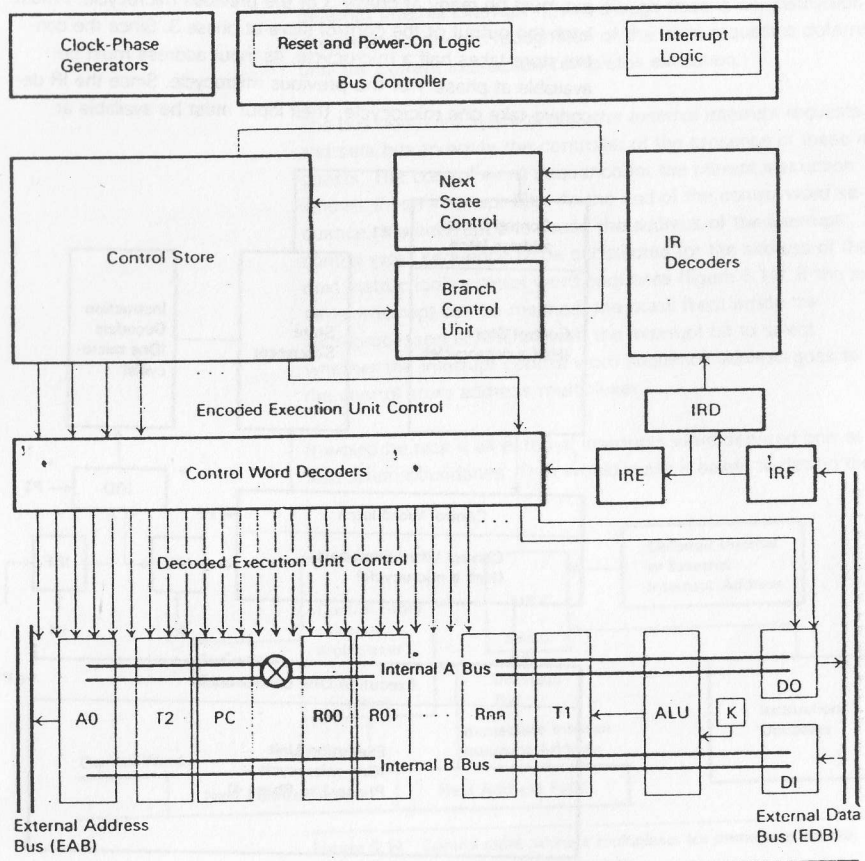


Figure 5.11 Microprocessor block diagram

to tell you how long the pieces take. The execution unit operates on a four-phase clock. A microcycle is one sequence of the four clock phases. The instruction decoders take one microcycle, and the control store takes half a microcycle. The control word decoders also take half a microcycle (see figure 5.12).

The execution unit operates in one microcycle and goes from phase 1 through phase 4. That means that the output of the control word decoders must be valid from the beginning of phase 1 through the end of phase 4, so I should latch it at phase 1. Since it takes two phases (half a microcycle) to operate, its input must be ready at phase 3 of the previous microcycle. I must latch the output of the control store at phase 3. Since the control store takes half a microcycle, its input address must be available at phase 1 of the previous microcycle. Since the IR decoders take one microcycle, their input must be available at

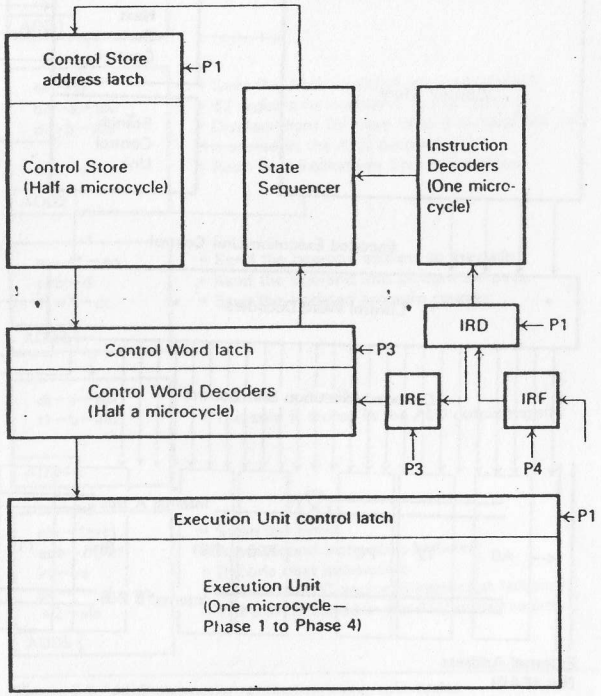


Figure 5.12 Microprocessor internal timing

phase 1—two microcycles before the execution unit is to operate. When I fetch an instruction, it is loaded into the IRF on phase 4. At the next phase 1, I could load the IRE and begin instruction decoding.

This presents more problems. Both the IRE and the control word are inputs to the control word decoders. The output of the control word decoders is saved at phase 1 (in a giant latch I call the execution unit control latch). Its inputs must be stable during the half microcycle before the end of phase 1. The output of the control store is okay because it is latched at phase 3. The IRE cannot change in phase 1, but it must be loaded at phase 1 to drive the IR decoders for a full microcycle. What do I do?

I had a problem with IRE anyway (see ADD5 in figure 5.10). In figure 5.11, I added the instruction register for decode (IRD) between IRF and IRE. IRD is there only to drive the instruction decoders, so it can be loaded anytime after IRF. It must be loaded at least two cycles before the end of the current control word sequence. (It can be loaded in the next-to-the-last state.) That allows one microcycle to decode the IRD value and one microcycle to retrieve the first control word and decode it. Figure 5.13 is the final ADD sequence.

Exceptions

I call the following "exceptions": interrupts, trace, bus error, reset, and power-on. There probably are others. Anything that isn't a valid instruction is an exception. This section tells what a microprocessor does with exceptions.

Interrupts

An interrupt is a request for a change in the normal instruction sequence. From the microprocessor's view, there are two kinds of interrupts. One kind comes from inside the chip (internal interrupts), and the other kind comes from outside the chip (external interrupts).

The following might cause an internal interrupt:

- fixed-point overflow
- divide-by-zero
- trace
- privilege exception (trying to do a supervisor instruction in user or problem state)

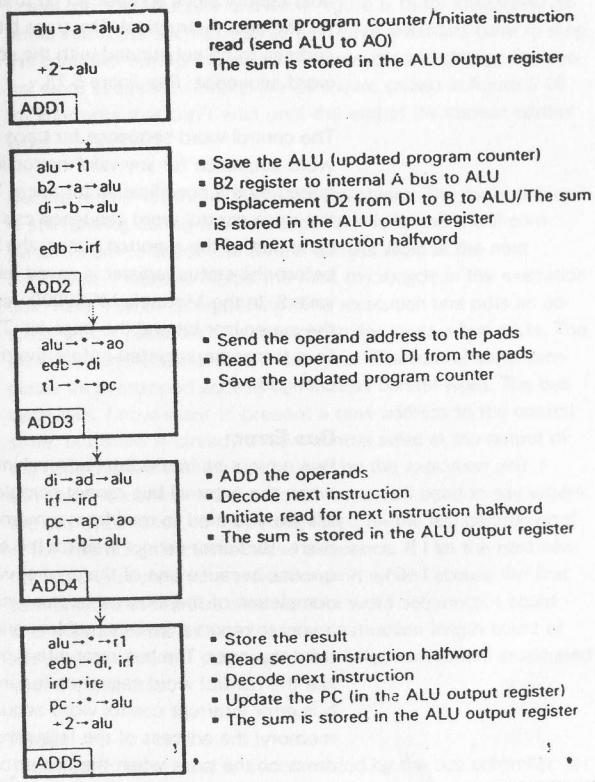


Figure 5.13 Final ADD sequence

- operation exception (trying to execute something that's not an instruction)

These conditions might cause an external interrupt:

- bus error
- peripheral (chip) service request
- reset request
- power-on
- test request

Because they are handled differently by the controller, I divide internal interrupts into two kinds: the kind that result from executing an instruction and the kind that do not. Arithmetic

overflow and divide-by-zero are internal interrupts that result from executing an instruction. Trace is an internal interrupt that does not result from executing an instruction.

Internal interrupts that result from executing an instruction are detected by the instruction's control word sequence. I do not need the hardware shown in figure 5.14 to deal with this but instead use a simple branch in the instruction's control word sequence. In this sense, these internal interrupts are handled immediately, so I call them immediate internal interrupts. An internal interrupt that does not result from executing an instruction is called a deferred internal interrupt. Trace is an example of a deferred internal interrupt. (Even though trace is on, instruction execution continues; recognition of the trace request is deferred until the current instruction completes execution.)

The bus controller synchronizes the external interrupt requests and sets bits to notify the controller of the presence of these requests. The control word sequence for the current instruction ignores these interrupt bits. At the end of the control word sequence, the interrupt bits cause the address of the interrupt control word sequence to be substituted for the address of the next instruction's control word sequence (figure 5.14). If the external interrupt can be masked, the mask (kept inside the microprocessor) is ANDed with the interrupt bit to select whether the interrupt control word sequence address goes to the control store address multiplexer.

It would be nice if all external interrupts were serviced only at instruction boundaries. That would make it easier to design the

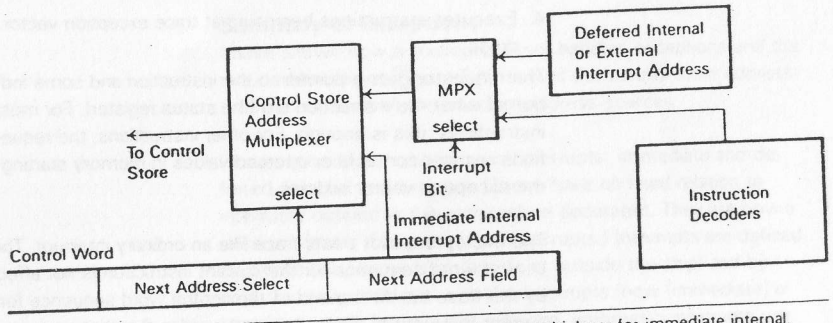


Figure 5.14 Control store address multiplexer for immediate internal, deferred internal, and external interrupts

controller and the interrupt control word sequence because the interrupts could be acknowledged when the internal state of the microprocessor was well defined. This works for what I call deferred external interrupts but not for bus errors, power-on (not much would be running with power-off), and some test and reset requests. These external interrupts I call immediate external interrupts.

Trace

To trace means to follow the trail of, to track down. A trace feature in a microprocessor offers a way to find out what a program is doing. A requester (program) turns on instruction tracing. This sets a trace bit in the microprocessor and causes a branch to the program being traced. The trace feature causes the traced program to be interrupted after one instruction, reports some internal state to the requester, and resets the trace bit. The requester analyzes the reported state (without tracing its own instructions) and turns on tracing for the next instruction in the program.

In the IBM Micro/370, trace is a System/370 program interruption with a special interruption code. The System/370 program interruption causes a program status word (PSW) swap and saves a 16-bit interruption code.

In the Motorola MC68000, trace does the following (after the current instruction completes execution):

1. Sets supervisor privilege state
2. Resets the trace bit
3. Pushes program counter and status register on supervisor stack
4. Executes instructions beginning at trace exception vector address

The requester gets a pointer to the instruction and some indication of what the instruction did (the status register). For most instructions, this is enough. For other instructions, the requester finds register contents or operand values in memory starting at the exception vector address.

The microprocessor treats trace like an ordinary interrupt. The control word sequence for the current instruction is not affected by the trace bit. At the end of the control word sequence for the current instruction, the last control word in the sequence at-

tempts to select the output of the instruction decoders as the next control store address (to go to the control word sequence for the next instruction). The trace bit causes the control store address to be substituted with the address of the trace control word sequence. See figure 5.15.

The control word sequence for trace will look like the control word sequence for any valid instruction. It will do the steps required by the specification for trace. The sequence of events in the trace control word sequence can be important. If the trace bit is part of the reported status, the trace bit cannot be reset before the status register is saved (either stored or temporarily saved). In the Motorola MC68000, reported status includes both the supervisor bit and the trace bit. The status register is saved (in a temporary register) before the trace and supervisor bits are changed.

Bus Error

Bus error is an immediate external interrupt. Bus error occurs when the external bus cannot complete an access request. Maybe you tried to read from a memory location that was not there. Bus error cannot wait for the end of the control word sequence because one of the control words is waiting for the completion of the external bus access. The bus error interrupt is a way to report a serious problem and to attempt to recover from the error. The bus error interrupt control word sequence is like the control word sequence for an ordinary instruction. The bus error interrupt control word sequence probably will store (in memory) the address of the failing instruction (PC) and the address on the pads when the bus error occurred.

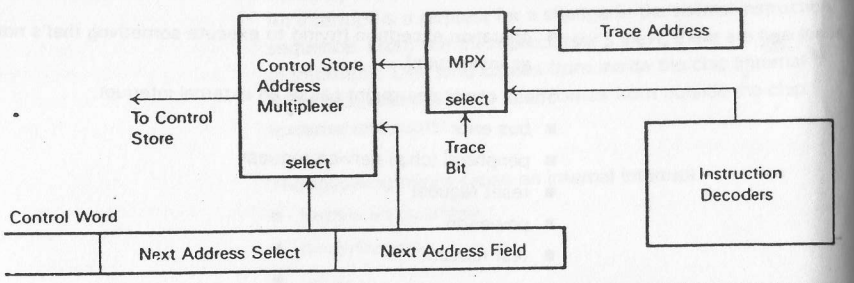


Figure 5.15 Control store address multiplexer for trace

I can't use the structure shown in figure 5.15 for immediate external interrupts such as bus error. These interrupts have to stop the controller during the current microcycle and change the control word sequence. I need the hardware shown in figure 5.16 for interrupts that can't wait until the end of the current control word sequence.

But we're still not there. Here's the problem: The execution unit is still running during the bus cycle that fails, and it will complete. If I am to substitute a new control word at the next microcycle, I have to inhibit the next microcycle in the execution unit. Here's what is happening: The execution unit puts an address on the pads, and the bus controller starts a bus cycle. The execution unit completes a microcycle. The control store completes its overlapped access to the next control word. The bus cycle fails. I now want to present a new address to the control store, but there is already a new control word at the output of the control store waiting to be used by the execution unit. I can't let this control word execute because I need to say where the bus cycle failed. It may have failed on the last control word in the instruction's control word sequence. If I let the next control word execute in the execution unit while I access the first control word of the bus error control word sequence, I could lose state information (such as the instruction length count or the program counter pointing to the failed instruction) associated with the bus error.

Reset

The request for a reset is synchronized by the bus controller. A reset is just like a bus error interrupt. I do not want to wait for the end of the current control word sequence. (I might be doing the reset because the hardware is stuck in an illegal state.) I want to stop doing the current instruction *immediately* and initialize the microprocessor. Reset differs from the bus error interrupt in that I do not have to preserve the internal state of the execution unit. I am going to set the internal state of the execution unit without reference to the current state. The reset interrupt control word sequence resets the status register and introduces a new program counter. It uses the same hardware as the immediate external interrupts (figure 5.16).

Power-On

Special circuits start the microprocessor after the power switch is turned on. The controller must come up in a known state.

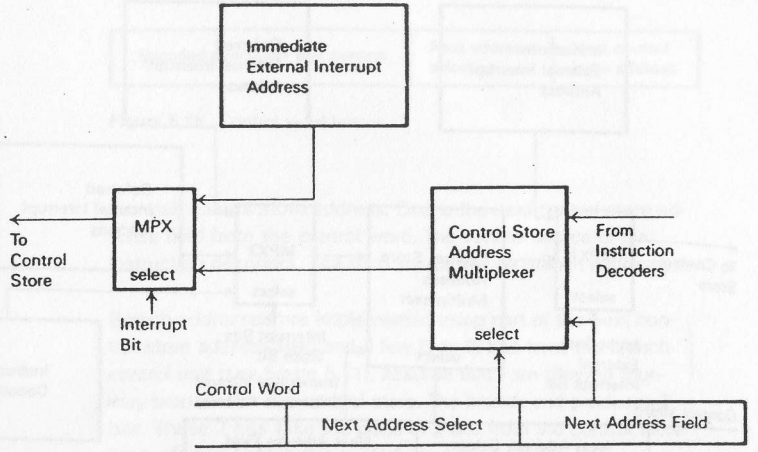


Figure 5.16 Control store address multiplexer for immediate external interrupts

then it should run the power-on control word sequence to start the microprocessor. The power-on control word sequence probably will initialize the status register and set the PC. It will have its own clocking and will use the same hardware as the immediate external interrupts.

Immediate external interrupts, reset, and power-on are not subject to masking by the programmer. These interrupts occur as soon as they are recognized—that is, immediately.

Summary of Exceptions

I have shown how a microprocessor handles exceptions and the required hardware. Figure 5.17 shows the control store address multiplexing with all the interrupt address sources.

I have named four categories of interrupts: immediate and deferred, internal and external. (These have no fixed relation to interrupts defined in the architecture document. They are how a microprocessor designer views interrupts.) Interrupts are defined by where they come from (inside or outside the chip) and by how the designer deals with the interrupts (now [immediate] or later [deferred]). Internal (for instance, arithmetic) interrupts use simple branches in the control word sequence. Deferred internal

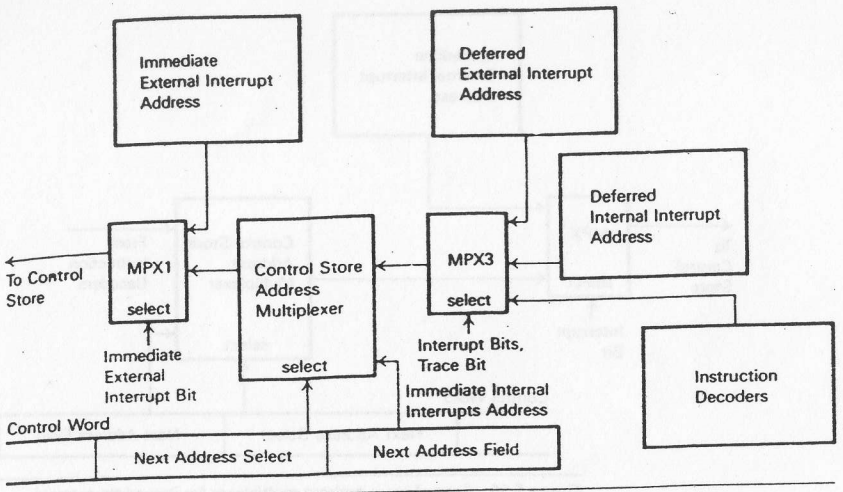


Figure 5.17 Control store address multiplexer for exceptions

and external interrupts substitute the address of the interrupt control word sequence in place of the pointer to the next instruction's control word sequence. Deferred internal and external interrupts are taken only at instruction boundaries. Immediate external interrupts substitute the address of the interrupt control word sequence for the address of the next control word and suspend action in the execution unit (while a new control word is accessed).

Previously, I defined an interrupt as a change in the normal order of executing instructions. But a designer does not necessarily care about whether instructions are executed out of the normal order. A designer cares about a change in the normal order of executing control words. For interrupts handled by a conditional branch within the control word sequence, there is really no change in the normal order of executing control words. The control word sequence just takes path A or path B. To a designer, immediate internal interrupts are not, in a sense, really interrupts. Deferred internal and deferred external interrupts are real to a designer because the normal order of executing control words is interrupted between control word sequences on a sequence boundary. Immediate external interrupts are real, too,

because the normal order of executing control words is interrupted within a control word sequence on a control word boundary.

Multiple Interrupts

What if more than one interrupt is present at one time? Let's suppose I am executing a Divide instruction with a zero divisor. The trace bit is set. A direct memory access (DMA) peripheral wants service (it is presenting an external interrupt request), and the operand access cannot complete. The winner of the contest is determined by the architect; the user's manual should rank the interrupts. Here is a typical ranking:

1. Immediate external interrupts
2. Immediate internal interrupts
3. Deferred internal interrupts
4. Deferred external interrupts

If the divide control word sequence cannot get the operands, it will not know there is a divide fault. The bus error interrupt generated by the failing operand access will outrank the arithmetic interrupt. In the absence of an immediate external interrupt, the arithmetic interrupt will rank highest because it is handled within the control word sequence. (The other interrupts have no indication that the instruction is not proceeding normally.) The internal interrupt (trace) ranks higher than the deferred external interrupt by controlling the hardware selection of the multiplexer (MPX3 in figure 5.17).

What if there are multiple interrupts of each kind? There is no problem with the immediate internal (arithmetic) interrupts because only one can occur per instruction. But there can be several peripheral chips that all want attention (the printer, the disk, the keyboard, the modem, and so on). There can be many deferred external interrupts, deferred internal interrupts, and immediate external interrupts. The interrupts in each category must be sorted out by hardware.

External interrupts are ranked by external hardware if there is only one pin for external interrupt requests. If there are several pins, as there are on Micro/370, the external interrupt requests must be ranked on the chip. All interrupts within a single category must be ranked. I use a PLA for each category to rank the

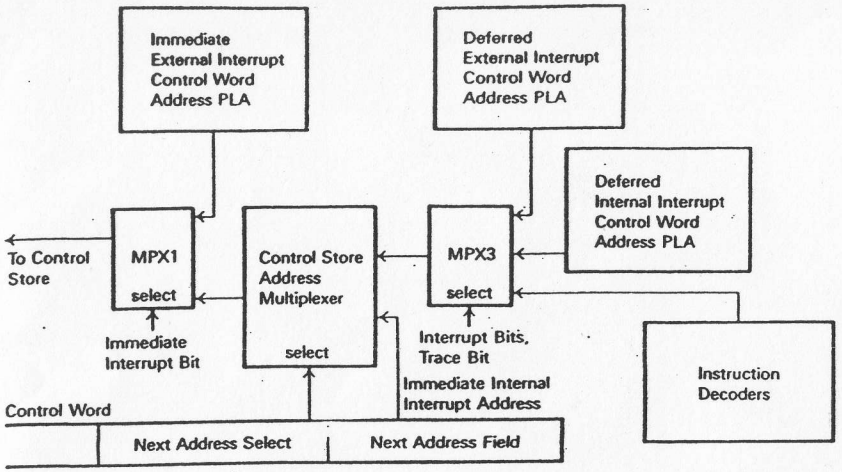


Figure 5.18 Control store address multiplexer hardware for exceptions

requests and to produce the address of the winning interrupt control word sequence. Figure 5.18 shows the hardware needed for the control store address multiplexer.

Control Store Address Selection

I am almost through with the control store address multiplexer. I need two more things: control store branching and additional addressing from the instruction decoders.

Branches

I have to have control word branches. Some of the instructions need them, and I need them to implement algorithms in control word sequences. I won't get a hardware divide unit on the chip this year, so I will have to implement divide with some kind of (looping) algorithm.

Figure 5.19 is the control word format. The field labeled "Next address selection" selects the next control store address for normal operation. (In the event of an exception, the highest ranking exception condition determines the next control store address source.) So far, I have shown only two sources for the

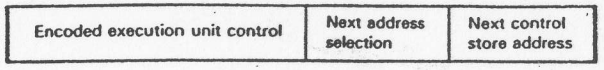


Figure 5.19 Control word format

(normal) control store address. One is the next control store address field from the control word. The second source is the instruction decoders. The third source will be branch addresses.

Branch addresses are implemented using part of the next control store address field and a few (1 to 3) bits from the branch control unit (see figure 5.11). Assume that I am allowed four-way branches in the control store. The branch unit produces 2 bits. These 2 bits take the place of 2 bits from the control store next address field in the control word (figure 5.20).

Instruction Decoding

Until now I have shown one address for instruction decoding, but you can use more. The Motorola MC68000 has about fourteen address modes. Most of the address modes can be used with any instruction. The control word sequences for calculating "effective address" are shared among instructions. A micro-

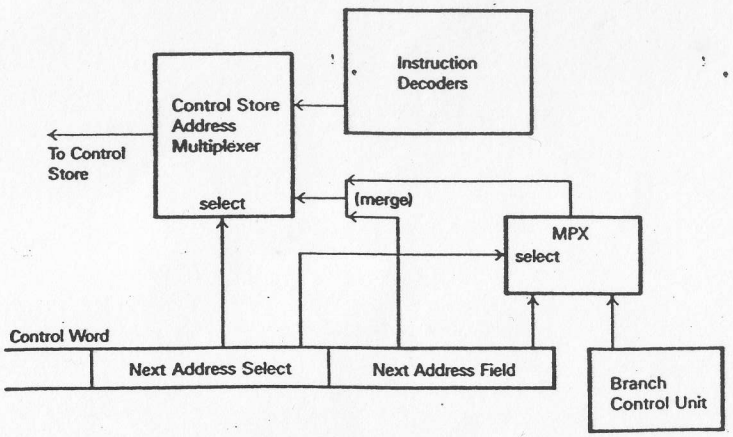


Figure 5.20 Control store address multiplexer for branches

processor can use at least two instruction decoders. (Call them A1 and A2.) One of the instruction decoders points to the address calculation control word sequence and the other points to the instruction control word sequence.

Micro/370 uses two instruction decoders. If there is no address calculation control word sequence, the first instruction decoder points to the instruction control word sequence, and the second instruction decoder is not used.

The Motorola MC68000 uses three instruction decoders, called A1, A2, and A3. There are three instruction decoding cases. Figure 5.21 shows which decoder points to which control word sequence for the three cases. (I lied a little. Physically, the Motorola MC68000 has only two instruction decoders, designated A1 and A2 A3. The A2 and A3 decoders are physically implemented by one PLA, which has one AND and two OR arrays.)

Operation, Specification, and Privilege Exceptions

I can put any bit pattern into the instruction decoders, but not all of them are legal op codes. The ones that are not legal cause an operation exception, which can be detected in the instruction decoder PLAs or with separate hardware.

A specification exception occurs when the op code is legal but something else is wrong with the instruction. In System/370, some instructions must designate an even-numbered register (they use an even-odd register pair for operands). If the programmer uses a multiply instruction with an odd-numbered register designated in the R1 field, a specification exception will occur.

	Control Word Sequence		
	Immediate Data or Address Calculation	Address Calculation	Instruction
Case 1			A1
Case 2		A1	A2
Case 3	A1	A2	A3

Figure 5.21 Instruction decoder operation

The instruction decoder points to the multiply control word sequence if the register designated by R1 is even. It points to the specification exception control word sequence if the register designated by R1 is odd.

Privilege exceptions can be implemented as deferred internal interrupts. If some instructions can be executed only in a supervisor state, the instruction decoders cough out the address of the instruction control word sequence if in supervisor state and the address of an exception sequence if not. If the supervisor state bit is included in the instruction decoding, you do not need a separate PLA to check for supervisor state.

Figure 5.22 is the control store address multiplexer for all control store address sources.

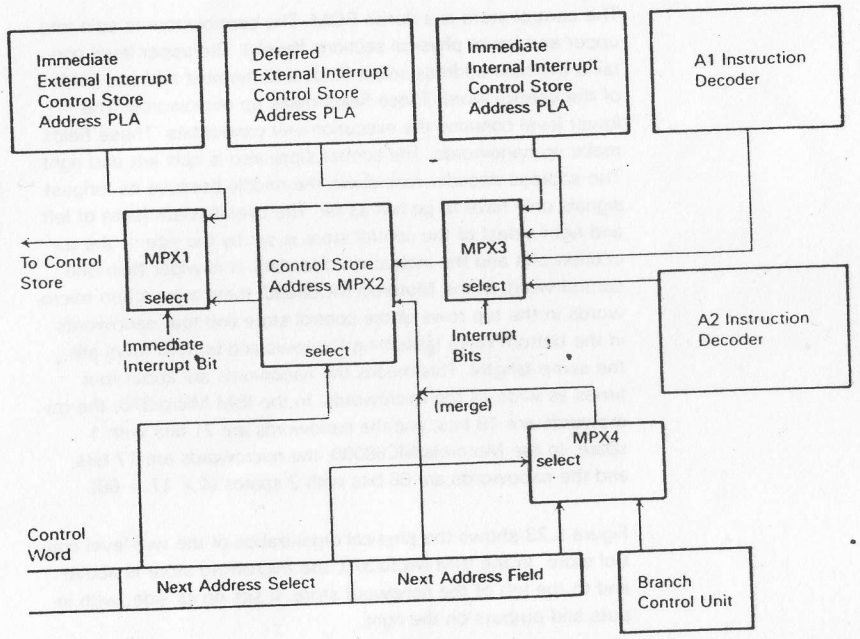


Figure 5.22 Control store address multiplexer for branches

Control Store

Operation

The address to the control store (the output of MPX1 in figure 5.22) is latched at phase 1. The control store takes two clock phases to operate. The output of the control store is latched at phase 3. The address enters the control store at the top and is decoded through the center of the control store. Both the next address select field and the next address field of the control word leave the control store at the top. There they go to the control store address multiplexer. They come out the top because most of the addresses for the control store are available at the top of the chip. There is more room there, and the lines do not interfere with the control lines to the execution unit. The control bits for the execution unit leave the control store at the bottom. They go directly into the control word decoders.

Organization

The control store is a dense ROM. The control store is split into upper and lower physical sections (levels). The upper level contains the next address select fields and the next address fields of the control word. These fields make up microwords. The lower level contains the execution unit control bits. These fields make up nanowords. The control store also is split left and right. The address decoder runs down the middle because its longest signals only have to go half as far. The overall width (sum of left and right sides) of the control store is set by the size of the execution unit and the instruction decoders. It is wider than one control word. In the Motorola MC68000, there are sixteen microwords in the top rows of the control store and four nanowords in the bottom rows (assuming top rows and bottom rows are the same length). This means the nanowords are about four times as wide as the microwords. In the IBM Micro/370, the microwords are 18 bits, and the nanowords are 71 bits with 1 spare. In the Motorola MC68000, the microwords are 17 bits, and the nanowords are 66 bits with 2 spares ($4 \times 17 = 68$).

Figure 5.23 shows the physical organization of the two-level control store. In the IBM Micro/370, the microword store is above and to the left of the nanoword store. It sits on its side, with inputs and outputs on the right.

In the Motorola MC68000, there is a 4-to-1 multiplexer at the bottom of the control store and a 16-to-1 multiplexer at the top.

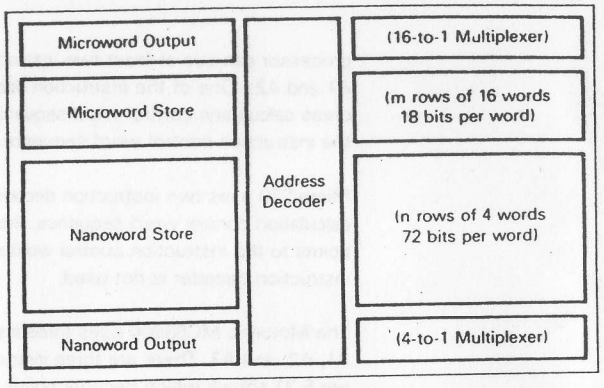


Figure 5.23 Two-level control store (organization)

In Micro 370, both control stores use a 4-to-1 multiplexer at the output. The bits of the words in each control store row are interleaved, so that when they exit the multiplexer, they are right next to where they are used. In Micro 370, the two branch bits are used to control the multiplexer selection because the multiplexer select bits are the last needed and the bits coming from the branch unit are the last available. The bits from the branch unit are on the critical path for the control unit. (Note: This configuration constrains placement of words in the control store.)

In both Micro/370 and the MC68000, the control store is implemented with a dynamic ROM. That allows two rows to share a ground line (normally, each row has its own ground line), which makes the circuits smaller. The ROM can be dynamic because the output is always latched once each microcycle. The output does not have to be static. By dynamic, I mean that for a fixed input, you see the output go to a value for a while and then go away. You have to latch it while it is valid. A static ROM maintains its output as long as the input remains unchanged (after some initial delay).

Nanoword Decoder

Nanoword Encoding

The nanoword is divided into fields. Each field controls one to a few resources in the execution unit. The nanoword fields are in the same physical order as the pieces of the execution unit. I

assign bits to each nanoword field according to how much it controls. Generally, I assign a separate nanoword field for each independent resource in the execution unit. If some pieces of the execution unit are never (or almost never) used at the same time, I might make them share a control field—but only if the controlled pieces are next to each other or if I can move them next to each other. If the pieces cannot be next to each other, sharing the nanoword field is not worth the effort because the decoded control lines would have to cross.

I would assign one nanoword field to the PC register, for example. I would then enumerate control events for the PC. The PC probably would drive the A bus or the B bus, or you could write to it from either the A or B bus. That's four control events. (If combinations of these control events occur in the same flowchart state [such as the PC driving the A bus and the B bus, for example], they become separate control events and you need more bits.) It looks as if I need 2 bits, but I really need 3 because I have to allow for the case that does nothing.

Nanoword Decoding

The nanoword decoder is shown in figure 5.24. Its inputs are the nanoword from the control store and the bits of IRE. Here is what the nanoword decoder does:

- decodes control store nanoword bit fields
- combines control signals of the nanoword fields and the IRE fields
- introduces timing signals to the logic terms
- geometrically aligns control signals with corresponding execution unit control points

Static Decoding

I call the decoders driven by the instruction register static decoders, because they do not change each microcycle. The fields of an instruction do not change during the execution of that instruction, but what you do because of the fields in the instruction does change from microcycle to microcycle. That's the stuff in the control store, and it must be decoded every microcycle. Static decoders make the instruction a fixed extension of every control word in that instruction's control word sequence. Op code, register fields, length fields, masks, and some ALU op codes are handled by static decoders.

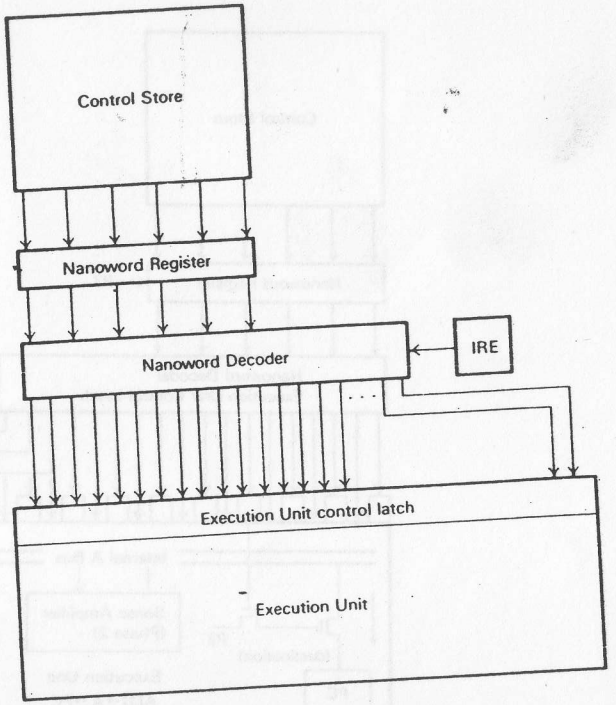


Figure 5.24 Microprocessor nanoword decoding

Timing

Timing is introduced to the execution unit control in the nanoword decoder. In the execution unit, source and destination transfers happen during clock phases 1 and 3, respectively. Since the nanoword is latched at phase 3, the source control signals can pass right through the execution unit control latches from the nanoword register. (Source control signals do not have to be latched.) The destination control signals must be latched at phase 1 to prevent hazards, because the nanoword is latched in the same clock phase (3) in which the destination control signals are driving the execution unit. Figure 5.25 shows how this works. Timing signals are uniformly introduced in the nanoword decoders, which reduces the chances for clock skew or hazards to cause logic errors.

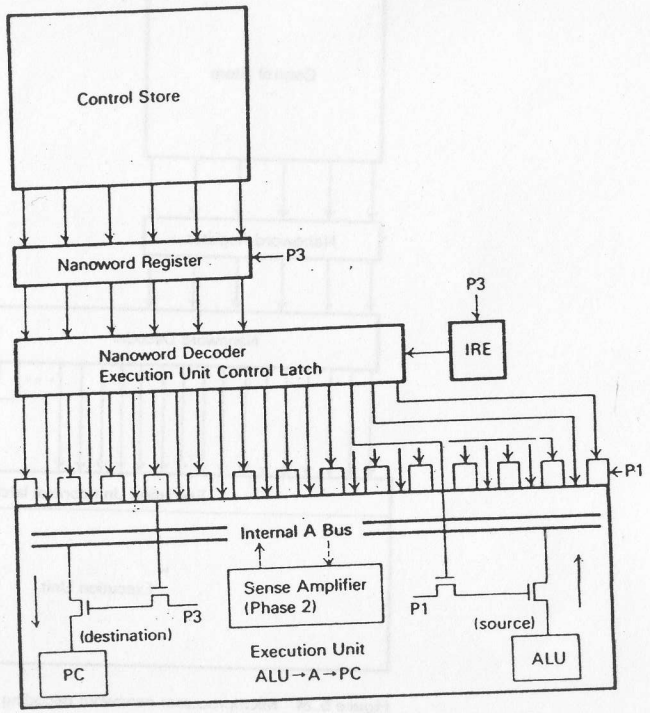


Figure 5.25 Microprocessor nanoword decoder, latch, and control

Decoding

Register Control An instruction such as Multiply Register takes several microcycles. Execution unit operations change with each microcycle, but the register pointers stay the same. There is no need to use a field in the nanoword to point to the register. I can let the register fields in the instruction select the registers and use a nanoword field to tell when and how I am using the selected registers.

As with everything else so far, it isn't that simple. The Multiply Register instruction in System/370 is an example. It specifies a register pair for the result, not just a source and a destination register. You have to check the instruction to be sure the R1 field specifies an even register. If it does not, you do not execute the instruction. If it does, you have to get the multiplicand from the odd register of the even-odd pair, multiply it by the contents of the register specified by the R2 field of the instruction, and put the result in the register pair specified by the R1 field. You can't just use the register pointers in the instruction register; you have to be able to force the low-order bit of the register pointer to be a one sometimes.

The architects of System/370 went even further, as some instructions have a base register field before the displacement. (There are two base-displacement fields for storage-to-storage instructions.) Worse, if the base register field (called B1 or B2) contains a zero, you do not use register zero but instead use the value zero. That is also true for the R2 field *sometimes*, which means that you have to look at the R1 field, the R2 field, and wherever the B1 and B2 fields are. Plus you have to be able to substitute a zero for register zero in some cases. And sometimes you have to force the low-order bit of the register pointer to be a one.

That isn't all. The System/370 architects defined instructions that write their results into specific registers. For example, the Translate and Test instruction (TRT) puts a byte in general register 2 and an address in general register 1. There is no field in TRT indicating general registers 1 and 2. There is an operand length field where register fields usually sit. Your register control hardware must be able to figure this out and point to registers directly.

The System/370 Load Multiple (LM) and Store Multiple (STM) instructions load or store (respectively) from one to sixteen general registers. The instructions only specify a beginning and ending register number. You have to load or store the registers in sequence beginning with the register specified by the R1 field and continuing (modulo 16) through the register specified by the R2 field. This means that you need hardware to generate sequential register pointers and to decide when to quit. Hardware to do this is shown in figure 5.26. You need one of these register control units for the A bus and another for the B bus. Even that isn't all there is to register control. Timing signals are introduced after the 5-to-17 decoder.

System/370 specifies a set of sixteen 32-bit general registers. In Micro/370, there is another set of sixteen 32-bit registers called

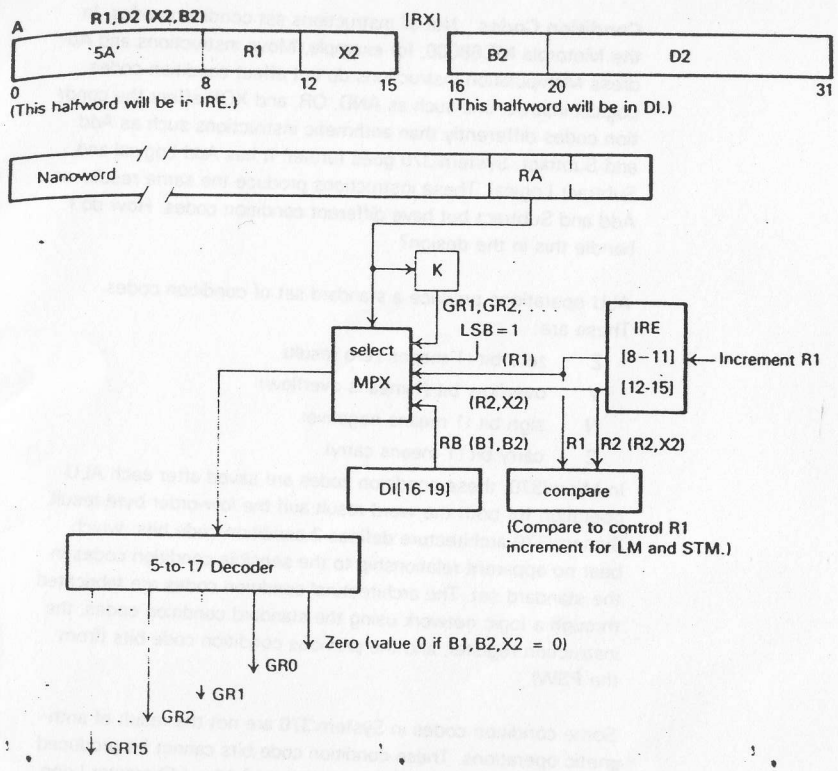


Figure 5.26 Register control decoder

shadow registers. A bit controls whether reading is from the general registers or from the shadow registers. Another bit controls whether writing is to the general registers or to both sets of registers. These signals are introduced after the 5-to-17 decoder.

ALU Function Control It appears that instructions such as OR Register can use the op code to select the ALU operation directly. This would save having to specify the ALU operation in the nanoword field. It might let you share the control word sequences for several instructions. (The control word sequences

for AND, ADD, OR, XOR, and SUB might differ by no more than the ALU operation.) But some instructions need more than one op code. Multiply, for example. I can't just have the ALU do a multiply (yet). I have to use adds and subtracts. Even simple instructions such as OR might have address calculations that need ALU operations other than one implied by the op code.

I do not want to give up the possibility of sharing similar control word sequences, but I need to be able to use several ALU operations in a single control word sequence. How can I do that? I use a table of ALU functions. The instruction register selects the row in the table, and the nanoword selects the column. Figure 5.27 is the ALU and condition code control table from Micro/370. (I will discuss condition codes shortly.)

As I wrote the flowchart for each System/370 instruction, I assigned it to a row in the ALU and condition code control table. The row in the table shows which ALU operations are used and which System/370 condition codes should be set. When I assigned an ALU operation, I had no idea how the System/370 condition code setting would be achieved, only how it must be assigned. I gave each System/370 condition code setting a letter of the alphabet. In a separate place, I wrote the equation corresponding to the letter for condition code setting (see figure 5.27). When the flowcharts were complete, I looked at the condition code equations I used and put them in a Karnaugh map. I tried to place them to minimize the PLA for implementation.

Several ALU operations are common to all instructions because you can never count on any instruction finishing normally. Some (instructions) contain illegal data. Some have forbidden formats. Some cause overflows. Sometimes the memory address is wrong or the memory system is broken. Lots of unusual things can happen. In System/370, for example, the architecture says specific things about what to do in each of these cases. The microprocessor cannot just quit. It has to try to recover. It stores the old state (PSW) and perhaps some information about what happened, reads a new PSW, checks it for validity, loads it, and begins running instructions at the location it specifies. All this has to be done potentially for any instruction in the IRE.

The common ADD and SUB columns are used by many instructions for address calculation. The AND and XOR columns are used by the special recovery and exception control word sequences for checking the format of the new PSW.

Column (Selected by Nanoword) 1	2	3	4	5	6	System/370 Instructions
A+B A	A+B 7	A+B 6	A−B A	AND A	XOR A	A,AH,AR,LTR,LNR, SRDA,SRA,SLA, SLDA
A+B A	A+B D	ADDC E	A−B A	AND A	XOR A	AL,ALR,DIAG
A+B A		A−B A	A−B A	AND A	XOR A	BXH
A+B A		B−A A	A−B A	AND A	XOR A	BXLE
A+B A	A−B 1		A−B A	AND A	XOR A	C,CR
A+B A		A−B C	A−B A	AND A	XOR A	CDS,CS
A+B A	A−B 9	OR A	A−B A	AND A	XOR A	CL,CLC,CLCL,CLI, CLM,CLR
A+B A	ADDC A		A−B A	AND A	XOR A	CVD
A+B A	B−A A	OR A	A−B A	AND A	XOR A	D,DR,EX,M, MH,MR
A+B A	A−B 7	ADDC E	A−B A	AND A	XOR A	DIAG,LCR,LPR,S, SH,SR
A+B A	A−B D		A−B A	AND A	XOR A	DIAG,SL,SLR
A+B A	A+B 5		A−B A	AND A	XOR A	ICM
A+B A	A−B 5		A−B A	AND A	XOR A	MVCL
A+B A	AND C	OR A	A−B A	AND A	XOR A	N,NC,NI,NR
A+B A	OR C		A−B A	AND A	XOR A	O,OC,OI,OR
A+B A	AND A	OR A	A−B A	AND A	XOR A	STNSM
A+B A	OR A	OR A	A−B A	AND A	XOR A	STOSM
A+B A	AND C	XOR B	A−B A	AND A	XOR A	TM
A+B A	A−B 9	XOR 8	A−B A	AND A	XOR A	TRT
A+B A	A+B 4		A−B A	AND A	XOR A	TS
A+B A	XOR C		A−B A	AND A	XOR A	X,XC,XI,XR

1. "A−B 7" means ALU operation is ADD, System 370 condition code is created by box 7 in the Karnaugh map below.
2. ADDC is A+B+1.
3. Unlisted instructions do not use column 2 or 3 operations.
4. DIAGnose appears on several lines—different forms.

Karnaugh Map (How System/370 condition codes come from the standard set of condition codes produced by the ALU)

cc0 cc1	0	1 $\bar{N}\cdot\bar{V}\cdot\bar{Z}+N\cdot V$ $N\cdot\bar{V}+\bar{N}\cdot V$	3	2
cc0 cc1	4 0 N	5 $\bar{N}\cdot\bar{V}\cdot\bar{Z}$ N	7 $\bar{N}\cdot\bar{Z}+V$ N	6 $cc0\cdot\bar{N}+$ $cc1\cdot\bar{N}+\bar{N}\cdot\bar{Z}$ N
cc0 cc1	C 0 $\bar{Z}$	D C $\bar{Z}$	F 0 $\bar{Z}$	E cc0 $\bar{Z}$
cc0 cc1	8 $\bar{Z}$ 0	9 $C\cdot\bar{Z}$ $\bar{C}$	B $Z\cdot cc1$ cc1	A cc0 cc1

(An empty box is a "don't care" for both bits.)

(cc0 keeps its old value in boxes A and E.)

Figure 5.27 Micro/370 ALU and condition code control table

Condition Codes Not all instructions set condition codes. In the Motorola MC68000, for example, Move instructions and Address Manipulation instructions do not affect condition codes. Logical instructions such as AND, OR, and XOR affect the condition codes differently than arithmetic instructions such as Add and Subtract. System/370 goes further. It has Add Logical and Subtract Logical. These instructions produce the same results as Add and Subtract but have different condition codes. How do I handle this in the design?

ALU operations produce a standard set of condition codes. These are:

- Z zero bit (1 means zero result)
- V overflow bit (1 means overflow)
- N sign bit (1 means negative)
- C carry bit (1 means carry)

In Micro/370, these condition codes are saved after each ALU operation for both the word result and the low-order byte result. System/370 architecture defines 2 condition code bits, which bear no apparent relationship to the sensible condition codes in the standard set. The architectural condition codes are fabricated through a logic network using the standard condition codes, the instruction register, and the previous condition code bits (from the PSW).

Some condition codes in System/370 are not the result of arithmetic operations. These condition code bits cannot be produced by the logic network. One example is the Move Character Long (MVCL) instruction, which sets the condition codes to indicate relative operand lengths—unless the operands overlap destructively. If there is destructive overlap, the condition code is supposed to be set to 3. Microcode detects destructive overlap cases using a variety of tests. If there is destructive overlap, I load a 3 directly into the PSW condition code bits using a constant from the execution unit. Condition codes that cannot be derived from the standard set are loaded directly using constants from the execution unit.

Figure 5.28 shows the logic for setting the System/370 condition codes in Micro 370. The op code from the IRE says whether the operation is a byte or word and how the condition codes are officially defined. An ALU and condition code control field (ACC) from the nanoword helps IRE control the ALU and condition codes. IRE gives static control, and ACC gives dynamic control.

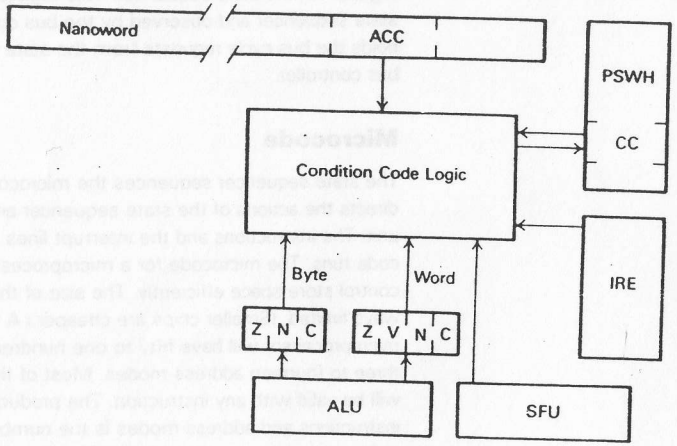


Figure 5.28 Micro/370 condition code control logic

Communication between Execution Unit and State Sequencer

I have already said that the state sequencer controls the execution unit. Now I will describe how the state sequencer knows what is going on in the execution unit.

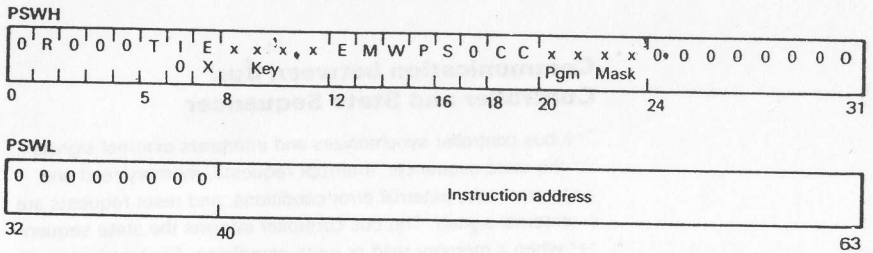
Status Register

The Status Register (SR) contains program-controlled state information. Usually this means condition code bits, interrupt masks, the PC (may be considered separately), and mode bits. In System/370, the SR is called the program status word (PSW), which actually is two words. Its format is shown in figure 5.29.

The Motorola MC68000 has the standard set of condition codes plus a special carry bit (x) for doing extended precision arithmetic. In System/370, the condition codes are 2 bits derived from the standard condition codes (and a few other things). Condition code bits are set implicitly by the programmer (as a side effect of executing an instruction defined to affect the condition codes). Micro 370 and the MC68000 have instructions that affect the condition codes explicitly.

The interrupt mask bits say which interrupts are allowed. In the Motorola MC68000, the interrupt mask is a level number. All interrupts are ranked. Interrupt requests with a level less than or equal to the current interrupt level are ignored. In System/370, one bit controls each interruption type. There is a bit for I/O interruptions, a bit for external interruptions, a bit for machine-check interruptions, and 4 bits for various program interruptions (see figure 5.29). In the MC68000, the interrupt mask is set to the level of the recognized interrupt. This prevents lower priority or equal priority interrupts from interfering with the interrupt service routine. Micro 370 loads a new PSW when it takes an interrupt, and this new PSW can prevent other interrupts from interfering with the service routine.

Mode bits control tracing, alternate register sets, instruction and data spaces, and privilege levels. Micro/370 has two sets of registers. Access for reading and writing is controlled by mode bits. Micro/370 also has a Control mode, a System/370 mode, and problem and supervisor states. Each of these is controlled by mode bits. (Some of the bits are not in the PSW in figure 5.29. They are not part of the architecture of System/370.) The MC68000 has bits to control tracing, instruction and data spaces, and supervisor and user states.



Program Status Word High (PSWH)

R	PER mask	(not used by Micro 370)
T	DAT mode	(not used by Micro 370)
IO	I/O mask	Enables I/O interruptions
EX	External mask	Enables external interruptions
Key	PSW key	Storage access key
E	EC mode	Controls PSW format
M	Machine-check mask	Enables machine-check interruptions
W	Wait state	1 = Wait 0 = Running
P	Problem state	1 = Problem 0 = Supervisor
CC	Condition code	Condition code bits
Pgm Mask	Program mask	Enable bits for program exceptions

Program Status Word Low (PSWL)

Instruction address
(Program counter)

Micro/370 uses 32 bits
System/370 uses 24 bits

Figure 5.29 System/370 PSW format

61

The Instruction Registers

In Micro 370, an instruction enters the chip through the data pads and is latched in the IRF. From the IRF, the instruction moves to the IRD, where it is decoded. The IRF is part of the execution unit, and the IRD is at the input to the instruction decoders. There is a 16-bit path from the IRF in the execution unit to the IRD in the state sequencer. The Motorola MC68000 works the same way, except the registers have different names.

Some instructions include operand length fields. The values from these fields must be sent to the execution unit for use in calculating operand addresses. The instruction moves to the IRE from the IRD just before execution begins. At the same time, the length field moves to a special function unit (SFU) in the execution unit. There is a 16-bit bidirectional path between the IRD and the SFU. There is also a 16-bit path from the IRE to the SFU.

Condition Codes

Condition codes are used in the state sequencer for branching. They can be stored in the execution unit or the state sequencer, as they are needed in both places. I put them in the state sequencer. In System 370, the condition codes are bits 18 and 19 of the PSW. The first word of the PSW contains a bunch of bits needed by the state sequencer, so I put the first word of the PSW in the state sequencer.

Communication between Bus Controller and State Sequencer

The bus controller synchronizes and interprets external signals for the state sequencer. Interrupt requests, memory read and write protocols, external error conditions, and reset requests are all external signals. The bus controller informs the state sequencer when a memory read or write completes. For termination of abnormal read and write bus cycles, the bus controller terminates the abnormal bus cycle and reports what happened to the state sequencer, through the Bus Status Register (BSR). There is an agreement between the designer of the state sequencer and the designer of the bus controller that says when the bus controller puts things in the BSR and how long the information stays valid. The BSR is controlled by the bus controller and is observed by the state sequencer.

The Processor Command Register (PCR) is the corresponding register for the state sequencer. This register is controlled by the state sequencer and observed by the bus controller. The PCR holds the bus cycle requests from the state sequencer to the bus controller.

Microcode

The state sequencer sequences the microcode, but microcode directs the actions of the state sequencer and the execution unit. The instructions and the interrupt lines control which microcode runs. The microcode for a microprocessor should use control store space efficiently. The size of the control store is always limited. (Smaller chips are cheaper.) A typical commercial microprocessor will have fifty to one hundred instructions and three to fourteen address modes. Most of the address modes will be valid with any instruction. The product of the number of instructions and address modes is the number of possible control word sequences. If a control word sequence is six or eight states, the number of states in the controller can be large.

Suppose I implement a separate control word sequence for each valid instruction. If the instruction is *w* bits long and all bit patterns are valid, the number of control word sequences is 2^w. If an average control word sequence is *m* microwords, then the control store needs $m*(2^w)$ states. For a 16-bit instruction, the controller will need about a million states. That is the upper bound. I don't need that many. I look at the op code and the address mode and don't worry about the register pointers. I let the register fields select the registers directly.

If there are *k* instructions and *a* address modes, then there could be k*a instruction sequences (if every address mode is valid with every op code). The number of states in the controller is k*a*m. The MC68000, for example, has about 50 instructions and about 14 address modes. Most instructions can be used with any address mode. If instruction sequences average 8 microwords, the controller needs 5,600 states. That is still large.

If the address calculation sequences are shared by the operation sequences, the controller needs only (k+a)*m states. I also might let the op code tell the ALU what to do. ADD, SUB, AND, OR, and XOR might all share the same control word sequence, for example. There is also some sharing of states at the end of

the control word sequences. The MC68000, for example, has only 544 states in the controller. Micro/370 has about 1,024 states in the controller (to do 102 IBM System/370 instructions).

Algorithms

The purpose of a controller is to translate a stimulus (instruction) into control signals for the execution unit. The more capability you put into the execution unit relative to the instruction set, the less complicated the controls are. If you simplify the instruction set enough, you can put everything into the execution unit. Voilà—no controller at all. If you have a multiplier in the execution unit, all the controller has to say is "multiply"—one signal.

The circuit designers would not give me a multiplier. Or a divider. Or a decimal ALU. The Motorola circuit designers would not give them to me for the MC68000, and the IBM circuit designers would not give them to me for Micro/370. Maybe next time I'll get the multiplier or the decimal ALU. But I won't get all I want. The more they give me, the more I add to the list of things I want. As long as you don't have all the hardware you need to do any instruction directly, you need control algorithms.

Neither Micro/370 nor the MC68000 has a multiplier, a divider, or a decimal ALU. Micro/370 does have a 64-bit shifter. The MC68000 has a 32-bit shifter. Multiplication, division, and the decimal operations are done on both microprocessors using simpler ALU operations. Multiplication and division use addition, subtraction, and shifting. The simple decimal Add and Subtract instructions use the binary ALU and a decimal correction factor. The decimal Add and Subtract instructions in Micro/370 are simple special instructions (the nine official IBM System/370 decimal instructions take a whole chip). The algorithms aren't even fancy. Micro/370 and the original MC68000 use a 1-bit Booth's algorithm for signed and unsigned multiplication. For divide, both Micro/370 and the MC68000 convert the operands to positive numbers. Then they divide using an unsigned 1-bit restoring-divide algorithm. Finally, they convert the answer and the remainder to the appropriate sign and check for overflow. Micro/370 also does the System/370 instructions Convert to Binary (CVB) and Convert to Decimal (CVD). These use simple shift and add algorithms.

Pretesting (IBM System/370)

The *IBM System/370 Principles of Operation* says that the CPU may not execute an instruction unless it can get all bytes of the

operand(s) that participate in instruction execution. You cannot store an answer unless you know, before you store the first byte, that you can store all the bytes. You cannot even change the System/370 condition codes unless you know the instruction will complete.

How does that affect the microprocessor? It means you have to figure out ahead of time which operand bytes will be used and make sure you can get to them. This is called pretesting. Pretesting is a requirement for the way virtual memory operation is defined in System/370. (You are not allowed to take page faults in the middle of most instructions.) The OR Character (OC) instruction, for example, reads two operands of length 1 to 256 bytes, ORs them, and stores the answer on top of one of the original operands. I do pretesting for OC by calculating the operand addresses and adding the length field to the result. Then I read the bytes at the end of each operand. If I can get both of them, I begin to execute the instruction from left to right. I have tested the right-most bytes first. Before I write anything in memory or change the System/370 condition codes, I will read the two left-most bytes (as the first part of the operation sequence). This works because page sizes are 2K-bytes or more in System/370.

That did not seem too painful. Consider the Add (A) instruction. That's really easy. The answer goes in a register, and the operand is only one word long. All you have to do is get the operand, add, put the answer in the register, and set the System/370 condition codes. If the operand memory access fails, nothing has been changed yet, so it's easy to handle the access exception.

Now consider the Store Characters under Mask (STCM) instruction. Bytes from a register are placed in contiguous locations in memory under control of a mask. Access exceptions are recognized only for the number of bytes specified by the mask. For pretesting, you have to figure out how many ones are in the mask and test that many bytes. The pretesting is almost as much work as instruction execution.

Trial Execution (IBM System/370)

For a few instructions, you cannot calculate the addresses for pretesting prior to instruction execution. The Translate (TR) instruction is one of those. (So is the Edit and Mark [EDMK] instruction, but Micro/370 does not do that one on-chip.) It is impossible to predict which bytes of Translate's second operand

will be accessed. And you cannot just test the complete address range of the possible second operand bytes because you are allowed to take an access exception only for operand bytes participating in instruction execution. The simplest way to test the operand bytes is to execute the instruction twice, once by trial execution. I practice executing the instruction—without changing any memory locations, registers, or condition codes. If everything goes well, I execute the instruction for real.

ALU Constants

There are not always two operands handed to you when you want to do an ALU operation. Sometimes you must increment the PC by the current instruction length. Sometimes you need a zero. Sometimes you must decrement an index register by one or by two. Where do these little constants come from? In Micro/370 and in the MC68000, the ALU "makes" them.

The ALU and the AU (arithmetic unit) each has a small constant generator attached to one of the inputs. In Micro/370, one of the ALU inputs always comes from the internal B bus, and the other input comes from the internal A bus or from the constant generator. The microcode decides which. The microcode also specifies the value of the constant. See figure 5.30.

	00	01	11	10
00	0 0→alu bd→alu	1 +1→alu bd→alu	3 +3→alu bd→alu	2 +2→alu bd→alu
01	4 +4→alu bd→alu	5 none	7 +7→alu bd→alu	6 +6→alu bd→alu
11	C −4→alu bd→alu	D −3→alu bd→alu	F −1→alu bd→alu	E −2→alu bd→alu
10	8 −8→alu bd→alu	9 −7→alu bd→alu	B ad→alu bd→alu	A −6→alu bd→alu

Transfer	Nanoword Bits
0→alu	0000
+1→alu	0001
+2→alu	0010
+3→alu	0011
+4→alu	0100
none	0101
+6→alu	0110
+7→alu	0111
−8→alu	1000
−7→alu	1001
−6→alu	1010
ad→alu	1011
−4→alu	1100
−3→alu	1101
−2→alu	1110
−1→alu	1111

Figure 5.30 Micro/370 microcode control for ALU constants

Constant	Nanoword Bits	Constant	Nanoword Bits
none	000000		
ag = 0	001010	ag = ba	100001
ag = 01	000001	ag = e8	011010
ag = 02	000010	ag = f0	101001
ag = 03	000011	ag = f2	101111
ag = 06	000110	ag = fa	101101
ag = 07	000111	ag = fe	101100
ag = 08	001000	ag = ff	100000
ag = 09	001001	ag = 100	011011
ag = 0c	001011	ag = 3f00	101011
ag = 0f	100110	ag = ff0f	101010
ag = 10	001100	ag = 80000	100111
ag = 20	001101	ag = 440000	101110
ag = 28	001110	ag = 3700400	011100
ag = 30	001111	ag = 66666666	011110
ag = 40	010001	ag = b1000000	100010
ag = 80	010011	ag = b2200000	100101
ag = 86	010110	ag = b80840ff	011111
ag = 88	010111	ag = ff000000	100011
ag = 8c	011000	ag = fff00000	100100
ag = b8	011001	ag = ffff8000	101000

Figure 5.31 Constants generated by Micro/370

The nanoword decoder to implement the control for the ALU constants is fairly simple. I chose the constants to be in the Karnaugh map so that I could just sign-extend the high-order bit of the field from the nanoword and get a small 32-bit constant. If the nanoword field contains 0101, then I prevent any ALU operation. If the nanoword field is 1011, then I select the internal A bus to the A side of the ALU. For all other cases, the sign-extended nanoword field goes to the A side of the ALU. What about numbers bigger than +7 or −8? More special cases.

Bus General Constants

Sometimes I needed bigger numbers. Sometimes I needed 32-bit masks. System/370 specifies where to store the old PSW and where to read the new PSW for a variety of exception conditions. The addresses are in real memory. Micro/370 has to "make" the addresses and check the new PSW format to be sure it is valid before the PSW is loaded. If it is not valid, it is a specification exception and requires another PSW swap. All this requires several constants and several masks. The masks and constants I used for Micro/370 are shown in figure 5.31.

Any microprocessor needs some constants for starting addresses on power-up and for interrupts. The MC68000 has several constants for interrupt vector addresses. There probably isn't another microprocessor that needs as many constants as Micro/370. Most microprocessor architectures do not have the protection and checking that are part of the architecture of a computer for large systems.

Internal Bus Control

In Micro/370 and the MC68000, two internal buses connect the resources of the execution unit. The buses lie on top of the macros composing the execution unit. Micro/370 has 32-bit internal buses, and the MC68000 has 16-bit internal buses. Operands and addresses travel from sources to destinations on these buses. This sounds easy, but the MC68000 uses 32-bit addresses and keeps a 32-bit program counter. Some instructions provide only a 12-bit displacement or a 16-bit operand. Some instructions manipulate bytes, and some manipulate 32-bit operands. How do I do all that with 16-bit buses? The 12-bit displacement can be sign-extended as it leaves the data input register. The 16-bit values are made 32 bits by dividing one bus into two segments. One segment propagates the 16-bit operand to the low-order half of the destination macro. The second segment copies the sign of the low-order half of the bus and duplicates it for all bits on the high-order bus. It propagates the sign to the high-order half of the macro.

Macros and internal buses in Micro/370 are 32 bits. Most operands are words (32 bits) or bytes. Displacements are 12 bits and must be zero-extended. There are some halfword operands and many byte operands. Micro/370 can zero-extend 8- and 12-bit operands leaving the Data Input register (DI). It can sign-extend halfwords leaving the DI.

When System/370 instructions load a byte into a register, the bytes not being written to must remain unchanged. The Insert Characters under Mask (ICM) instruction loads from 0 to 4 bytes into a general register. Any bytes not being loaded must remain unchanged. I had to be able to store bytes in a general register under control of a mask. I also needed explicit control of which bytes got loaded. Load Address (LA), for example, loads the low-order three bytes and leaves the high-order byte unaffected. Insert Character (IC) puts its operand in the low-order byte of the general register and leaves the other bytes unchanged.

I control the bytes on the bus by having separate bus sense amps for each byte. If the byte sense amp does not turn on, the corresponding byte in the destination macro is unaffected. The buses are true and complement buses. The true and complement rails are both precharged high. If the sense amps do not copy the source information onto the bus, then when the destination macro is connected to the bus, the macro does not lose its value. The microcode controls the sense amps. I can vector the instruction mask field to the sense amp control hardware, or I can specify the byte sense amps in the microcode directly. The default value is for all four byte sense amps to work on each bus.

Bus Interface

The on-chip bus controller is the interface between the (synchronous) internal state sequencer and execution unit and the (analog) external world.

Synchronization

Synchronization is difficult. An interrupt request looks like an analog signal to the bus controller. It looks analog (even though the sending device views the signal as digital) because the bus controller does not know when the signal changes. No big deal. You just sample it (copy it into a register on each occurrence of some clock edge). There is a finite probability that you copied the signal while it was changing. Your latch might not resolve the signal before you check it. Get a faster latch. Maybe there was noise on the line, and you copied a spike. There is no interrupt request, but you think there is. The faster the latch, the more likely you are to copy a spike. Get a slower latch. Now your time to resolve the signal is longer—increasing the probability that you will not resolve it in time. Get a fast latch and compare the sampled value with the current value on the request line. This is better but more expensive and slower. And still you could have an intermediate state.

All that is just for one external signal line. The bus controller looks at a bunch of lines. Micro/370 has 106 signal pads, and all the asynchronous input-request lines must be sampled and resolved. The bus controller decides which requests are valid and synchronizes them for the state sequencer.

External Bus Control

The bus controller runs the bus protocol for talking to external devices (such as the memory and coprocessors). The micro-

processor is the king of the bus. It grants bus cycles to other devices that request bus cycles.

I did not design the bus protocol or the bus controller for either Micro/370 or the MC68000. I'm glad I didn't; it would have been too difficult.

Mode Control

Micro/370 does not do all the instructions defined by the *IBM System/370 Principles of Operation.* (No 370s do.) Because of this, I need a way to re-create the missing parts of the architecture. Micro/370 has a feature called Dual mode. An internal bit called the mode bit divides all operations into two categories: Control and 370. User programs run in 370 mode in 370 space. If the microprocessor encounters an instruction it does not recognize, there is a PSW swap for an operation exception. The PSW swap causes Micro/370 to change to Control mode. Control mode has a separate address space, called control space, with access to 370 space for reading and writing. In Control mode, a program can emulate missing instructions using the available 102-instruction set and adjust locations in 370 space to have the appearance of having executed the missing instruction. The program running in 370 space never knows that the instruction was not executed directly by the microprocessor.

The mode bit cannot be seen or explicitly changed by a program running in 370 space. A program running in control space changes the mode bit with a special load PSW instruction. After the Load PSW, the program resumes in the 370 space.

Shadow Registers

Micro/370 has two sets of general registers because a group wanting to use the registers for emulation of instructions asked us to include them. Both sets are used by the programs in 370 space. One set looks like the general registers and is used for loads and stores. The other set, the shadow registers, keeps a copy of the general registers. Programs running in 370 memory have no control over the shadow registers. They cannot even tell that the extra registers are there. When an exception condition changes the mode to Control mode, the program running in control memory has explicit control over the use of the general registers and the shadow registers. The program can select the following modes:

1. Read from shadow registers; write to general registers
2. Read from general registers; write to general registers
3. Read from general registers; write to general registers and shadow registers (default)

If you are emulating an instruction for a program running in 370 space, you probably will do your work using mode 2. When you are done, you will return the general registers to the values expected by the program running in 370 space using mode 1. The programs running in 370 space will use mode 3 so that there is always a set of saved registers whenever the mode changes to control space.

Special Instructions

Micro/370 has some instructions that are not completely specified by the *IBM System/370 Principles of Operation.* Those instructions all have DIAG (Diagnose) op code '83' hex. The Diagnose instruction is a privileged System/370 instruction that performs model-dependent functions. It can help the program in control space emulate the instructions Micro/370 does not implement. For example, it provides primitive decimal Add and Subtract instructions so the program in control space can emulate the official System/370 decimal instructions more easily. There is a Calculate Effective Address instruction that looks at an instruction's op code and calculates the effective address(es) for the operands. There are instructions that help diagnostics and fake various interrupts; dump the contents of the execution unit; and explicitly control operand read and write spaces (whether operands are accessed in 370 space or control space) and operation of the register sets. None of these instructions operates in 370 mode.

C H A P T E R

6

The IBM Micro/370 Microprocessor

Micro/370 is a single-chip microprocessor. It comprises a clock phase generator, a bus controller, an execution unit, and a processor controller. I specified the execution unit and designed the processor controller. The execution unit represents the programmer's picture of System/370 because it contains the general registers and the PSW. The processor controller translates the official System/370 instruction bit patterns into the appropriate sequence of operation commands for the execution unit and the computer's memory. The bus controller makes the chip work in an electrical environment with memory chips, interrupts, and real-time events. The bus controller synchronizes signals for the processor controller.

Micro/370 is a square chip 10mm on a side. It has peripheral pads (see figure 6.1). To the designers, the chip has a definite orientation: The execution unit is on the bottom, and the bus controller is on the top. Instruction and data addresses go out on the pads around the bottom left corner. The external data bus (EDB) is on the pads around the lower right corner. Bus controls and interrupts occupy the pads going around the top of the chip.

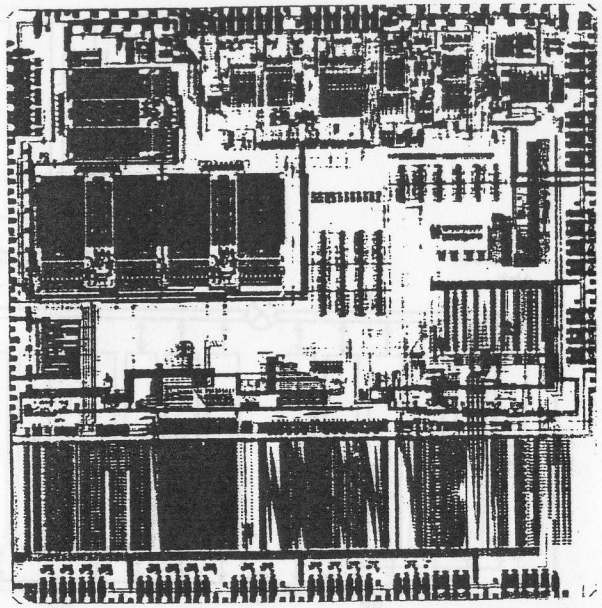


Figure 6.1 Micro/370 chip

Execution Unit

The execution unit is shown in figure 6.2. Notice the bus couplers between AT and AG on the internal A and B buses. These bus couplers are like relays; they divide the buses into separate sections. The bus couplers allow you to transfer data all across the bus or divide the bus into sections for concurrent transfers within sections. The main advantage is the ability to calculate instruction and data addresses using the left section of the bus and simultaneously do arithmetic and data transfers using the right section. But the left section can still use the data pads occasionally, and the right section can use the address pads.

Let's tour the Micro/370 execution unit from left to right:

AO—the 32-bit address output buffer. It consists of the drivers for the address pads. AO gets the addresses from either the A or B bus or directly from AU.

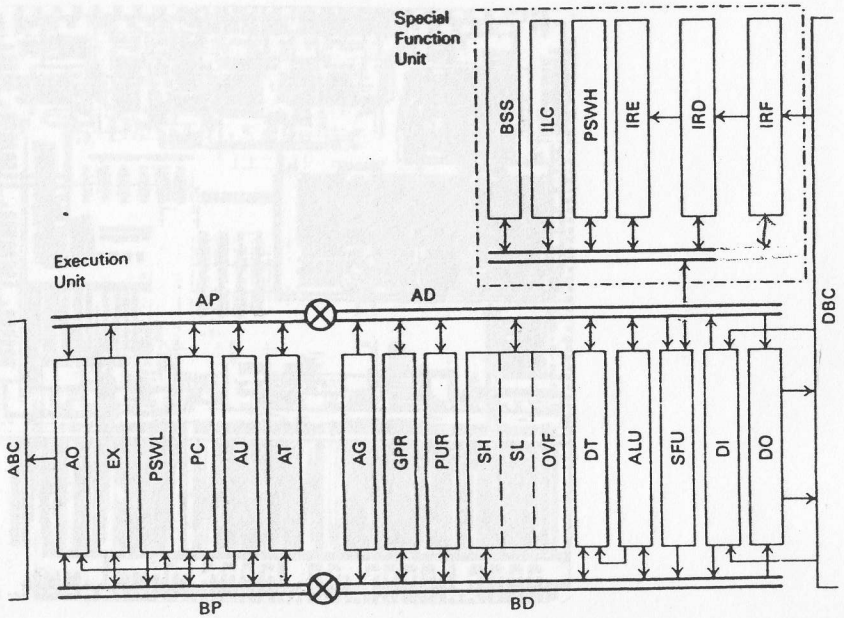


Figure 6.2 Micro/370 execution unit and special function unit

EX—the Execute register. This register remembers the original instruction address during an Execute instruction. It has a read path from the A bus and a bidirectional path to the B bus.

PSWL—the low word of the System/370 PSW. It can send data to the B bus and is automatically loaded from the AU between instructions.

PC—the program counter holds instruction addresses. It has bidirectional connections to the A and B internal buses and a write-only connection from the AU.

AU—the 32-bit arithmetic unit. The arithmetic unit, which only does two's-complement add and subtract, is used primarily for address calculation and instruction address update. Its output is connected (optionally) to either the A or B bus, AO, PSWL, and PC. One input to the AU comes from the A bus. The other input can come from either the B bus or a constant generator. The AU has a result register at its output.

AT—an Address Temporary register connected to the A and B buses.

Bus couplers and sense amplifiers—the sense amplifiers. There is a set for each section of each bus. The individual bytes of the sense amps for the data sections of the bus can be controlled with a mask.

AG—the address generator. This unit sits above the execution unit but has a 32-bit path connecting it to the A or B bus. AG is the source for all the special constants and addresses required by Micro/370.

GPR—the sixteen 32-bit general registers and the sixteen 32-bit shadow registers. The sixteen general registers are R0 through RF. The shadow registers are a copy of the general registers and are called S0 through SF.

PUR—a special-purpose unit used to aid execution of instructions that move nibbles around. (MVN, MVZ, MVO, PACK, and UNPK are examples.) The Pack-Unpack Register (PUR) is connected to the A and B buses, but what happens when an operand arrives at PUR depends on both the bus from which it comes and the instruction that sent the operand. PUR includes an 8-bit register.

SH/SL/OVF—the 64-bit shifter. It does single and double shifts from 0 to 63 bits in one cycle. SH is the high-order half and is connected only to the B bus. SL is the low-order half and is connected only to the A bus.

DT—a Data Temporary register connected to the A and B buses. The ALU can write directly to DT.

ALU—the arithmetic and logic unit. The ALU is used for all general-purpose arithmetic. It can write to the A or B bus or directly to DT. One input to the ALU always comes from the B bus, and the other comes either from a constant generator or the A bus. The ALU has a Result register at its output.

SFU—the special function unit. Here is where I lumped all the odd things I needed to do, including decimal digit and sign checking; the path from the instruction register to the execution unit (needed for getting operand length counts, masks, and immediate values); the path from the bus controller to the execution unit; the path from the state sequencer to the execution unit; the path to the PSW (high-order word); and nibble swaps. The A bus can write to SFU, which can send operands to the B bus. SFU includes a 32-bit register.

DI—the 32-bit Data Input register. Its output can be connected to the A or B bus.

DO—the 32-bit Data Output register. Its input can come from the A or B bus.

BSS—the 8-bit communication register between the state sequencer and the execution unit.

ILC—the instruction length count. ILC holds the length of the instruction being executed. It is located above the execution unit and connected via the internal C bus to the SFU.

PSWH—the high word of the PSW. It is located above the execution unit and connected via the internal C bus to the SFU.

IRE—the instruction register for execution. It holds the first halfword of the instruction being executed and is located above the execution unit and connected via the internal C bus to the SFU.

IRD—the instruction register for decode. It holds the first halfword of the next instruction so decode can begin during execution of the current instruction. It is located above the execution unit and connected via the internal C bus to the SFU.

IRF—the instruction register for fetch. It usually holds the second halfword following the beginning of the current instruction. It is located above the execution unit and connected via the internal C bus to the SFU.

Figure 6.3 is the block diagram of Micro/370. Notice the strong similarity between the block diagram and the actual chip (figure 6.1). This similarity is characteristic of the flowchart method.

Control Store Organization

The Micro/370 control store is divided into two parts. The top part holds microwords that control the state sequencer. I call this the micro control store. The bottom part holds nanowords that control the execution unit. I call this part the nano control store. The control store address is always the same for both parts of the control store. The microword exits the right side of its control store, and the nanoword exits the bottom of its control store (figure 6.4). The control store is just a ROM. The control store address is 10 bits, for 1,024 possible words. Each control store address produces one microword and one nanoword. Microwords are 18 bits. The micro control store is organized physically into two banks of 1,024 10-bit sections. This gives one 20-bit microword each access (18 bits of microword and 2 spares). Nanowords are 71 bits. The nano control store is organized physically into 4 banks of 1,024 18-bit sections. This gives one 72-bit nanoword each access (71 bits of nanoword locations and 1 spare). The control store implements 1,024 nanoword

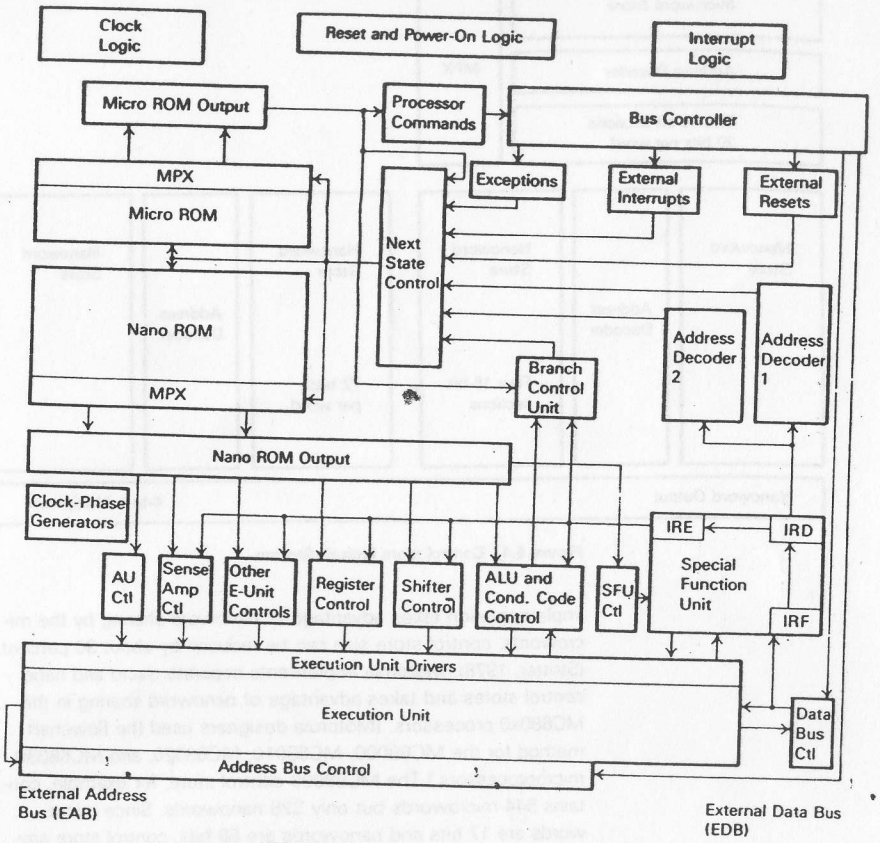


Figure 6.3 Micro/370 block diagram

locations and 1,024 microword locations, but only 984 microword and 984 nanoword locations are occupied (40 control store locations are empty).

The flowcharts for Micro/370 contain 899 microwords and 543 nanowords. There are fewer nanowords than microwords because the flowcharts allow sharing of nanowords. (Nanowords are the tasks in a flowchart box, and the microwords are the rest—sequencing and external bus control.) If the control store

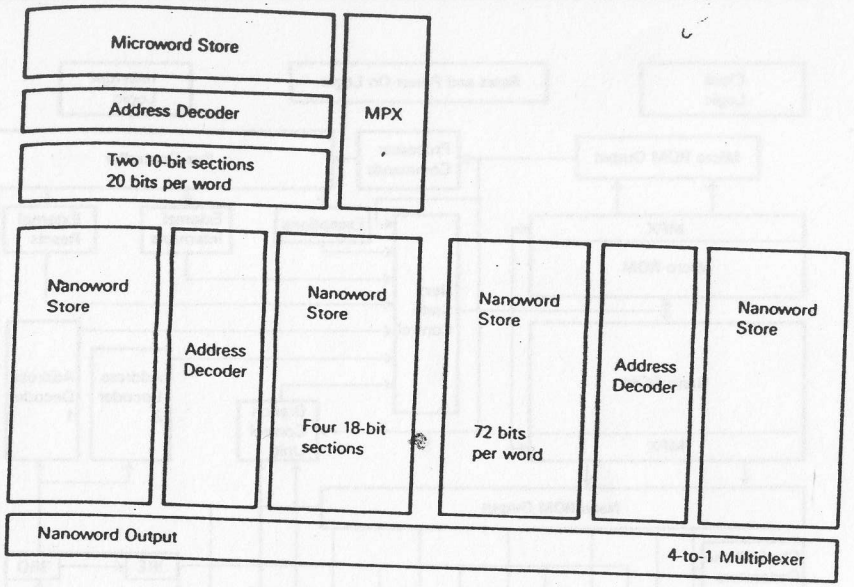


Figure 6.4 Control store logical diagram

implementation takes advantage of nanoword sharing by the microwords, control store size can be reduced by about 30 percent (Stritter, 1978). Motorola implements separate micro and nano control stores and takes advantage of nanoword sharing in the MC680x0 processors. (Motorola designers used the flowchart method for the MC68000, MC68010, MC68020, and MC68030 microprocessors.) The MC68000 control store, for example, contains 544 microwords but only 328 nanowords. Since microwords are 17 bits and nanowords are 68 bits, control store savings is about 32 percent. Micro/370 separates the micro and nano control stores but implements the complete 10-bit address range for each control store. Since the control store address is 10 bits, each control store contains 1,024 words.

Two paragraphs back, I said that there were 984 control words (and 40 empty locations) in each Micro/370 control store. In the next paragraph, I said that there were 899 microwords in the flowcharts. It seems there should be only 899 control words in the control store. There would be only 899 control words if control word addresses were independent. For microcode branches, 2 bits from the branch control unit substitute for the 2 low-order

control store address bits. Microcode branch targets, therefore, must have related addresses. If the same microword is the target of more than one microcode branch, the control word may have to be duplicated in the control store.

Decoders

A1 and A2 are the instruction decoders. They are just above the execution unit and to the right of the control store. The instruction decoders change the official System/370 instruction op codes into addresses for microsequences in the control store. The input to these decoders comes from instruction accesses, which pass through the external data bus connected to the pads around the lower right corner of the chip. The output of these decoders is an address for the control store.

A1, the larger PLA, provides the address for the first microsequence. A2 provides the address for the second microsequence, if any. An instruction with an operand address calculation may have a pointer to the address calculation microsequence in A1 and a pointer to the operation microsequence in A2. A simple register-to-register instruction will have a pointer to the operation sequence in A1 and will be a "don't care" for the A2 PLA.

Next Address Control

The source of the next address to the control store is (usually) determined by the current microword. The microword might provide its own next address. It also might choose an address from A1 or A2. In the event of a microcode branch, the microword will provide a partial address and name a branch condition. The branch unit will provide the last 2 bits of the address. The last microword in an instruction will choose A1—which should be the address of the first microword in the microsequence for the next instruction. If an interrupt is pending, the interrupt will preempt the A1 decoder and provide an address to the interrupt processing microsequence. (This allows interrupts only on instruction boundaries.) A bus error condition or an external reset signal can preempt any other next address selection.

Interface to Bus Controller

The state sequencer is synchronous, and the outside world is not. The bus controller is the interface between the two. The designer of the bus controller and the designer of the state se-

quencer agreed on how they would communicate. In Micro/370, the state sequencer talks to the bus controller through the Processor Command Register (PCR). The state sequencer has to know when to change PCR, and the bus controller has to know when to look at PCR. The bus controller presents status, interrupts, and reset conditions to the state sequencer in the Status Register, the Interrupt Request Register (IRR), and the External Request Register (XRR), respectively. The bus controller knows when it can change these registers, and the state sequencer knows when it is okay to look at the registers. The PCR and Status Register tell when the address and data buses are active.

Figure 6.5 is my view of how Micro/370 looks in a system. To me, everything that is attached to Micro/370 looks like some kind of memory. The state sequencer has to be able to communicate with each of the different types. Sometimes the communication has to be explicit. For example, loading the page tables or the storage keys requires direct communication with the translation buffer or the storage key memory. Sometimes the communication is implicit. ROM, for example, is never addressed explicitly but can be anywhere. (ROM must be there somewhere to provide the starting program on power-up.)

The access indicator field in the microword holds encoded information for the bus controller. It tells the bus controller the following:

- access width (byte, halfword, or word)
- read or write
- service cycle or memory cycle
- instruction or operand
- real address space
- read from write space
- halt
- reset
- no access

When I wrote the flowcharts, I used access indicators as necessary until I had more than sixteen. I thought sixteen would be plenty, but it was not. I had allocated 4 bits in the microword for the access indicator. I changed it to 5. Even that did not seem to be enough. There were too many kinds of reads: reads from instruction space, reads from write space, reads from real memory, and reads from main memory. Each of these could be word or halfword reads, and operand reads could be byte reads.

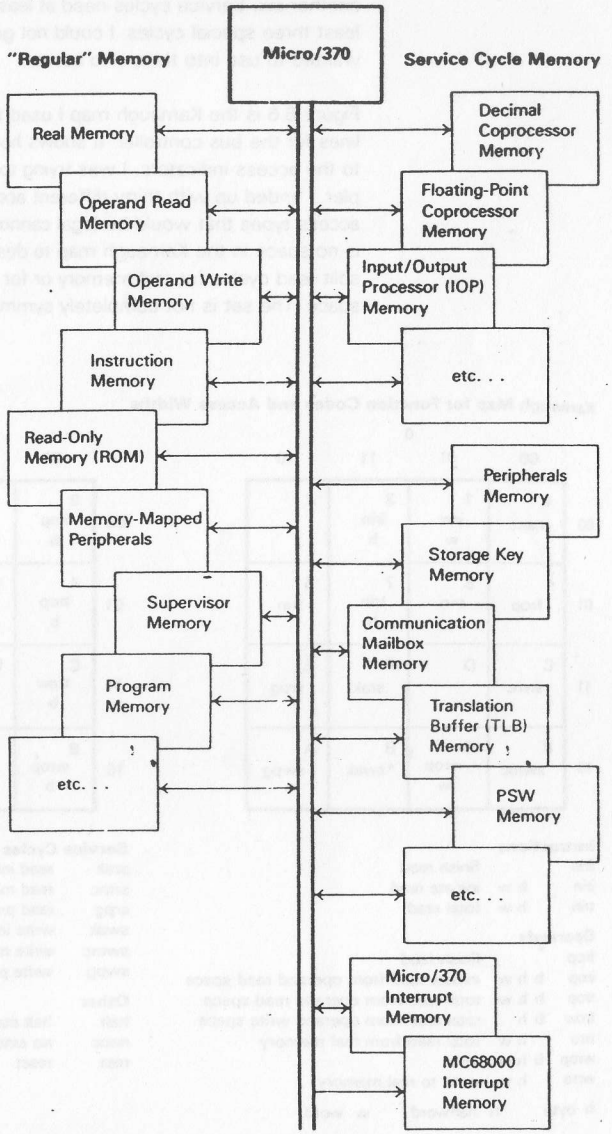


Figure 6.5 Micro/370 "regular" and service cycle memory

These reads could use twenty-four codes. The writes could use another six. Service cycles need at least six, and there are at least three special cycles. I could not get all the access types I wanted to use into thirty-two cases.

Figure 6.6 is the Karnaugh map I used to implement the control lines for the bus controller. It shows how I assigned the values to the access indicators. I was trying to make the decoder simpler. I ended up with thirty different access indicators. Some access types that would be legal cannot be used because there is no space in the Karnaugh map to describe them. There are no split read cycles for real memory or for reading from the write space. The set is not completely symmetrical. At first I used

Karnaugh Map for Function Codes and Access Widths

0

	00	01	11	10
00	0 nacc	1 irin w	3 irin h	2
01	4 frop	5 trin w	7 trin h	6 frin
11	C srmc	D	F srak	E srpg
10	8 swmc	9 wrop w	B swak	A swpg

1

	00	01	11	10
00	0 irop b	1 irop w	3 irop h	2 halt
01	4 trop b	5 trop w	7 trop h	6 rest
11	C trow b	D trro w	F trro h	E trow h
10	8 wrop b	9 wrro w	B wrro h	A wrop h

Instructions

frin		finish read
irin	h w	initiate read
trin	h w	total read

Operands

frop		finish read
irop	b h w	initiate read from operand read space
trop	b h w	total read from operand read space
trow	b h	total read from operand write space
trro	h w	total read from real memory
wrop	b h w	write
wrro	h w	write to real memory

b byte h halfword w word

Service Cycles

srak	read interrupt acknowledge
srmc	read microcode control
srpg	read program control
swak	write interrupt acknowledge
swmc	write microcode control
swpg	write program control

Other

halt	halt state
nacc	no external operation
rest	reset

Figure 6.6 Microword function codes and access widths

whatever I wanted in the flowcharts. Then it came time to make it all fit in the control store. I had to throw out some cases. There were only one or two occurrences of things such as split cycle reads from real memory or from the write space, so I threw them out.

Memory Access Bus Codes

I have a rule of thumb that says I can have two microcycles for every bus access. In Micro/370, I can start a bus cycle during the first microcycle and finish it on the second. States that initiate a bus cycle but do not wait for the result are called initiate cycles. Those that do not initiate the access but must wait for the result are called finish cycles. Those that initiate the bus access and wait for the result are called total cycles.

FRIN—finish read of an instruction. The FRIN cycle does not supply an address to the bus controller. It waits for only one result, since it does not depend on access length.

IRIN—initiate read of an instruction. The IRIN state puts the address on the bus to initiate the instruction read. The state sequencer does not wait for the memory to respond but goes immediately to the next microinstruction, which will have a FRIN function code. IRIN is valid for word and halfword accesses. (There are no byte-length instructions in System/370.)

TRIN—total read of an instruction. The TRIN state sends the address to the bus controller and "hangs" until the instruction is fetched. TRIN is valid for word and halfword accesses.

FROP—finish read of an operand. FROP reads from the current operand read space and is otherwise the same as FRIN.

IROP—initiate read of an operand. The same as IRIN, IROP reads from the current operand read space. IROP is valid for byte, halfword, and word.

TROP—total read of an operand. TROP reads from the current operand read space and is valid for byte, halfword, and word.

TROW—total read of an operand from operand write space. Some instructions read and write the operands (XC, OC, NC, for example). Micro/370 can have separate address spaces for read and write operands. If an operand is read and write, where do you get it (the read space or the write space?) and where do you put it? Brion Shimamoto (my associate in the Micro/370 project) decided that we should get it from the write space and write it to the write space. Even if the operand is write only, you must have a way to read from the write space to be able to test to see whether the location is there before you try to write to it.

TROW permits reading operands from the write space and pretesting write locations. It is valid for byte and halfword.

TRRO—total read of an operand from real memory. TRRO is valid for halfword and word. "Real" memory is assigned special values by the System/370 architecture. In addition, real memory's page is always available, and there are no byte operands in real memory. So I didn't have to do any byte reads from real memory for operands or pretesting. There also are no instructions in real memory (that's why the function code is an operand read). If I didn't need the function code, I didn't assign it.

WROP—total write of an operand. WROP writes to the current operand write space and is valid for byte, halfword, and word.

WRRO—total write to real memory. WRRO is valid for halfwords and words. (There are no byte operands in real memory.)

Instruction accesses use five function codes (five positions in figure 6.6). Function codes appear at the pins. Operand accesses use sixteen. Of those, eleven are for various reads, and five are for writes.

Service (Bus) Cycles

Service cycles (designated Sxxx) are bus cycles used to communicate with all the special devices (e.g., translation buffer, storage key memory, coprocessors, interrupting devices) attached to the Micro/370 bus. Function codes for any service cycle will be 111. For most service cycles, the low-order three address bus bits A[3-1] will be zero. If A[3-1] are zero, the bus cycle is a sense (read) or control (write) service cycle. Sense or control is indicated by the R/W pin. If A[3-1] are not zero, the bus cycle is a Motorola interrupt acknowledge cycle. The MC68000 interrupt acknowledge protocol puts the interrupt level on address bit A[3-1]. How do you know whether you are talking to the storage key memory or to the translation buffer? Micro/370 uses the data strobes—word strobe (WS), upper data strobe (UDS), and lower data strobe (LDS). It is okay to use them because all service cycles are word operations, so they can use the strobes to separate the devices. For this use, the strobe pins WS, UDS, and LDS are called Service Address 3, 2, and 1 (SA3, SA2, SA1) respectively. Any of the data transfer acknowledge signals (BTACK, DTACK, WTACK, or VPA) is a valid response. The flowcharts distinguish the following types of service cycles. These types set the pins to the values listed in table 6.1.

SRAK—service cycle read interrupt acknowledge. SRAK is used to read an interruption code from an interrupting device.

Table 6.1 Sense or Control Service Cycles—Pins

Functions Codes = 111				
R/W	SA3	SA2	Sa1	
x	0	0	0	not used—you cannot tell it is there
x	0	0	1	reserved for programmers using Diagnose
x	0	1	0	reserved for programmers using Diagnose
0	0	1	1	write to coprocessor
1	0	1	1	read from coprocessor
0	1	0	0	not assigned
1	1	0	0	interrupt acknowledge low word
0	1	0	1	new PSW broadcast
1	1	0	1	interrupt acknowledge high word
0	1	1	0	purge translation buffer
1	1	1	0	not assigned
x	1	1	1	storage key memory

SRMC—service cycle read under microcode control. The microcode uses SRMC to read from and signal the special devices attached to the Micro/370 bus.

SRPG—service cycle read under program control. The programmer uses SRPG to read from and signal the special devices attached to the Micro/370 bus. The SRPG service cycle cannot fail. There is no invalid response (even bus error is okay). The response is encoded and passed to the program in general register 1 and the condition codes. The strobes and the address are set by the address calculated in the Diagnose sense service cycle. Program-controlled service cycles allow the programmer to communicate with and control external hardware, such as the translation buffer and the storage key memory.

SWAK—service cycle write. SWAK is used to broadcast the new PSW anytime significant bits change (other than just the condition codes or the instruction address). The high-order word of the PSW is placed on the address pads (its low-order byte is all zeros, so it does not look like a Motorola interrupt acknowledge cycle, despite its acronym). The low-order word of the PSW (the instruction address) is placed on the data pads. (The PSW is always in System/370 extended control [EC] mode.)

SWMC—service cycle write under microcode control. The microcode uses SWMC to write to and signal the special devices attached to the Micro/370 bus.

SWPG—service cycle write under program control. The programmer uses SWPG to write to and signal the special devices

attached to the Micro/370 bus. The SWPG service cycle cannot fail. There is no invalid response (even bus error is okay). The response is encoded and passed to the program in general register 1 and the condition codes. The strobes and the address are set by the address calculated in the Diagnose control service cycle. The programmer could even broadcast a fake PSW, for example.

HALT—request to the bus controller to drive the halt pin (pad).

NACC—no external bus access requested.

REST—request to the bus controller to drive the reset pin (pad).

Execution Overlap

In Micro/370, the execution of the current instruction is overlapped with decoding of the following instruction and the prefetch of the second following instruction. This is true for both microinstructions and instructions. (I call this overlap instead of pipelining because there is still only one instruction in execution at a time.) It is the job of each instruction microsequence to make sure that the next instruction has been fetched sufficiently in advance to allow decoding. The current microsequence also has to fill IRF with the halfword following the next instruction. This allows back-to-back execution of halfword instructions. Inside the chip, while the current microinstruction is being executed, the following microinstruction is being read from the control store and decoded. Fetch, decode, and execute are overlapped at the instruction level. At the microinstruction level, fetching and decoding are overlapped with execution.

Prefetching

It is common for a microprocessor to assume that the next instruction is there, get it, and then throw it away if there is a branch. The MC68000 does this. If you are doing a prefetch while you are decoding a branch instruction, you do not even know you have a branch instruction yet because you are executing the instruction before the branch instruction. How could you know whether the branch will be successful? You can't. You just do the prefetch, and if the instruction (or the memory location or whatever) is not there, the instruction blows up. See figure 6.7.

Your microprocessor is overlapped for fetching, decoding, and execution. While you are executing the Add Register (AR)

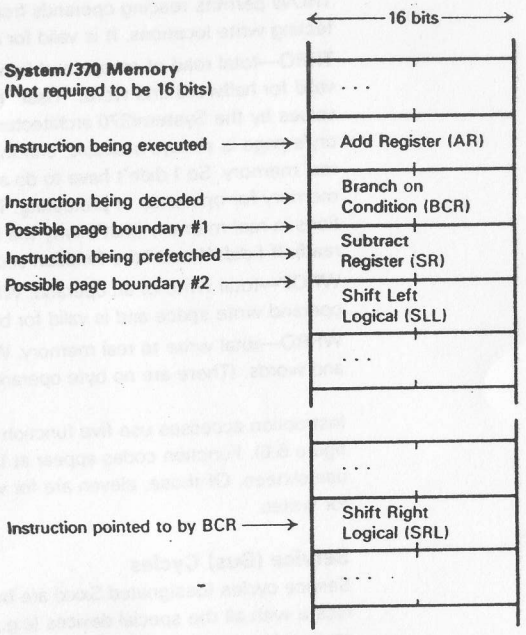


Figure 6.7 How instructions overlap in Micro/370

instruction, you are decoding the Branch on Condition Register (BCR) and fetching the Subtract Register (SR) instruction. That is what the MC68000 does. It is also what Micro/370 does. Further, once the Add Register instruction is done, you begin to execute the BCR. You have three choices for accessing the instruction stream:

1. You can check the branch condition and then access the proper next instruction location.
2. You can access the next sequential instruction since you already have its address, then throw it away if you don't need it.
3. You can assume the branch will be taken, calculate the branch address, and access the instruction at the branch destination. Throw it away if you don't need it.

Notice that the SR instruction is already in the CPU when you begin to execute the branch instruction. It was fetched by the

AR instruction microsequence and is now being decoded. In figure 6.7, I am talking about whether to fetch the shift left logical (SLL) or the shift right logical (SRL) instruction. In the MC68000, I assumed that branches would be taken about 60 percent of the time, so I calculated the destination address and began the prefetch there. Then I threw out whichever instruction I didn't need. (I had both the SR and the SRL.) You have to do another access before you leave BCR because you have to access the halfword following the next instruction. Even if you did not do the extra prefetch, you would have to delay one cycle whenever you branch—to load the next instruction (in this example, SRL) into IRD and decode it. In Micro/370, I used options 2 and 3. Sometimes I got the branch destination instruction, and sometimes I got the next sequential instruction. I got the branch destination instruction for BCR and Branch on Count (BCT), for example. For Branch on Condition (BC), I got the next sequential instruction while I was calculating the branch destination address.

Execution overlap is a good idea, but it is not free. System/370 architecture says: "If you don't need the instruction, it doesn't have to be in memory." On a branch instruction, you are not allowed to take an access exception on the next sequential instruction (because it isn't there) if the branch is taken. "But it's the bus controller that recognizes the access exceptions," you say. Yes, it is. But it's the state sequencer that decides what to do about access exceptions. I designed the state sequencer. In the Motorola MC68000, all the access exceptions were treated alike—the access blew up with a bus error. In Micro/370, all access exceptions are not alike. The bus controller reports the outcome of any bus cycle to the state sequencer in the Status Register. The Status Register is 17 bits—one for each way a bus cycle can terminate. The bits are mutually exclusive. The bits in the Status Register are divided into two groups, normal terminations and exceptional terminations.

For normal terminations, there is no problem. For exceptional terminations, the action of the state sequencer depends on the nature of the exceptional termination. If the exceptional termination results from an operand access (either read or write), then the access is part of the current instruction and you branch to the appropriate exception processing microsequence. If the exceptional termination results from an instruction access, you do not do any exception processing. Instead, you encode the exception condition in a byte and save it.

If the exception occurred when you were loading IRF, save the encoded exception byte in a register called IFX. If the exception occurred when you were loading DI, save the encoded exception byte in a register called DIX. If you were loading both IRF and DI, load the exception byte in both IFX and DIX. If you subsequently do an access that does not cause an exception, erase the byte in the corresponding exception register. When you move the contents of the IRF into IRD to decode the next instruction, you also should move the contents of IFX into IDX. If you try to do an A1 call (using the output of the instruction decoder, which is driven by IRD), or you try to load IRE from IRD and the IDX register is not zero (that is, it contains an exception code), then branch immediately to the exception sequence for the encoded exception condition. If you try to use the contents of DI and the value in DIX is not zero, then branch immediately to the exception sequence for the encoded exception condition.

Figure 6.8 shows how the prefetch protection registers are set for the instruction example of figure 6.7. The three instructions

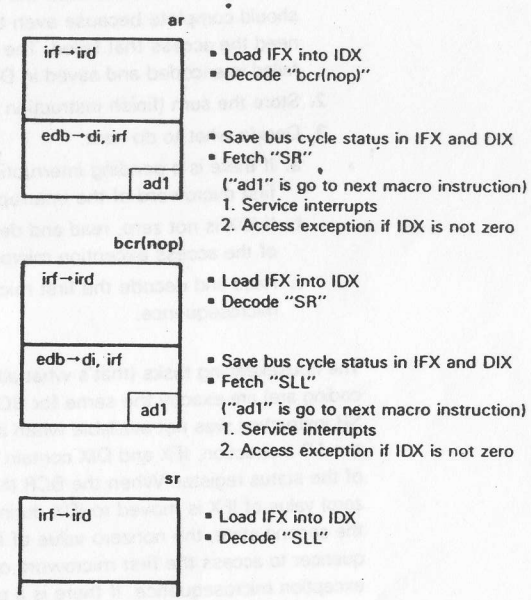


Figure 6.8 Simple instruction overlap in Micro/370

(AR, BCR, and SR) are halfword instructions. If the R2 field of the BCR instruction is a zero, the instruction becomes a no operation (NOP). In figure 6.7, I threw out all the tasks not related to the prefetch protection registers. This sequence of macro instructions illustrates how instruction fetch, decode, and execution are overlapped. Each of the instructions executes in two states. (Rule of thumb: The on-chip state sequencer is twice as fast as the external bus.) During the two states to execute an instruction, a bus access to the halfword following the next instruction is in progress. Here is the process:

A. During the first cycle of AR
   1. Start the external bus cycle to read the SR instruction.
   2. Send BCR (NOP) to the instruction decoder (and decode).
   3. Move IFX to IDX.
   4. Add the operands (begin instruction execution).

B. During the second cycle of AR
   1. Finish the external bus cycle (read the SR instruction into DI and IRF). If this instruction access cannot complete, the bus controller will tell the state sequencer why (via the Status Register). The AR instruction should complete successfully. In this case, even the BCR (NOP) instruction should complete because even that instruction does not need the access that failed. The reason the bus cycle failed is encoded and saved in DIX and IFX.
   2. Store the sum (finish instruction execution).
   3. Decide what to do next.
      a. If there is a pending interruption, read and decode the first microword of the interrupt microsequence.
      b. If IDX is not zero, read and decode the first microword of the access exception microsequence.
      c. Read and decode the first microword of the BCR (NOP) microsequence.

The housekeeping tasks (that's what all the prefetching and decoding are) are exactly the same for BCR (NOP) and SR. If the SR instruction was not available when it was accessed during the AR instruction, IFX and DIX contain the (nonzero) encoding of the status register. When the BCR (NOP) executes, the (nonzero) value of IFX is moved to IDX during the first state. During the second state, the nonzero value of IDX causes the state sequencer to access the first microword of the appropriate access exception microsequence. If there is a pending interruption from

outside the chip, it is given preference. Pending interruptions outrank page faults and other access exceptions. (You wouldn't want to fix the page fault and then take an interrupt and have the interrupt code kick your new page out of memory.) Both AR and BCR (NOP) complete even though an access exception occurred during the AR and before BCR (NOP).

The sequence of figure 6.8 is a good performance indicator for the state sequencer. It shows the processor running back-to-back minimum length instructions. There should be a good balance among bus cycle time, microword access and decode, instruction decode, and execution time. In this example, there is. The bus cycle time is twice the other times. But instruction decode and microword access and decode are sequential, and most instructions take two cycles (compute, then store result). Fetch, decode, and execute are balanced.

Figure 6.9 illustrates how the prefetch protection registers are set and used for instructions longer than one halfword. I have been talking about register-to-register instructions, which are only a halfword long, do not do memory accesses for operands,

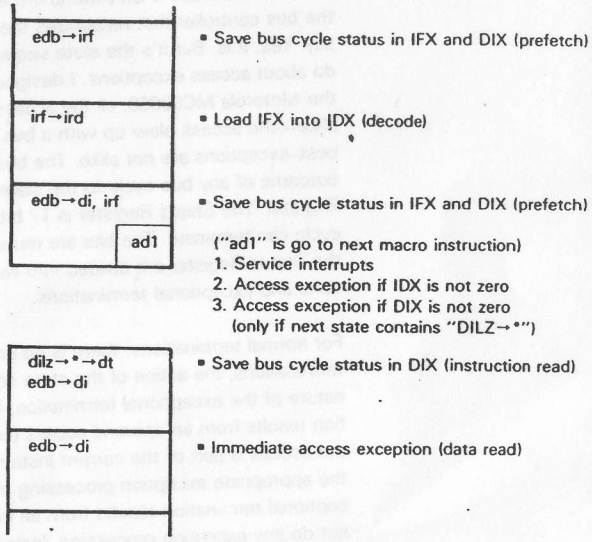


Figure 6.9 Instruction overlap in Micro/370

and do not calculate operand addresses. If the external bus access fails for any operand read or write, you do not have to save anything (in a prefetch protection register). You cannot complete the current instruction, so you branch to the microcode for the appropriate access exception as soon as you recognize the condition.

Look at the last state in figure 6.9. It is an operand access. If the access fails, you want the next microword executed to be the first microword in the exception microsequence. That's not easy. The external bus access does not complete until the end of the state, but in the interim, the state sequencer is accessing and decoding the next microword in the instruction's microsequence. If the current access fails, you do not want to execute the next microword in the sequence because it might change the state of a register the programmer can see (such as a general register or the PSW). That would violate the requirement of the architecture that an instruction does not affect the programmer's state unless it can complete.

If there is an access exception, the state sequencer issues a signal called "euhold" (execution unit hold) to the execution unit. The euhold signal causes the execution unit to ignore the following microword. (It may do the register reads and internal bus transfers, but it will not change any of the internal registers.) In addition, each time the bus controller returns an access exception status, it ignores the subsequent bus command. (Since the microword accesses are overlapped, the next bus command is issued before the end of the current command and must be ignored if the current command fails.) This one-microcycle "hiccup" by the bus controller allows the state sequencer to issue a new command. If the access exception occurs on a prefetch, the state sequencer saves the encoded status and reissues the original next command. This scheme allows prefetching access exceptions without stopping the current instruction execution. When a prefetch access exception occurs, there is a one-microcycle "hiccup" (one state) in instruction execution time.

The last instruction access in any microsequence loads both IRF and DI (see, for example, figures 6.8 and 6.9). The halfword being loaded is the second halfword following the current instruction. If the next instruction is only one halfword, then the instruction following the next instruction is in IRF. If the next instruction is two or three halfwords long, then the base register

designator and displacement for operand address calculation are in DI. If the current state loads an access exception code in DIX, the state sequencer checks the next microword (the one being decoded) and each subsequent state to see whether the contents of DI will be used. If the execution unit tries to use the contents of DI, the state sequencer will change the next state to the beginning of the exception sequence. This ensures that the exception occurs during the instruction that attempts to use the missing data (as opposed to the instruction that attempted to get the data).

Figure 6.10 is the block diagram of the address decoders controlled by the prefetch protection registers. It shows a simplified

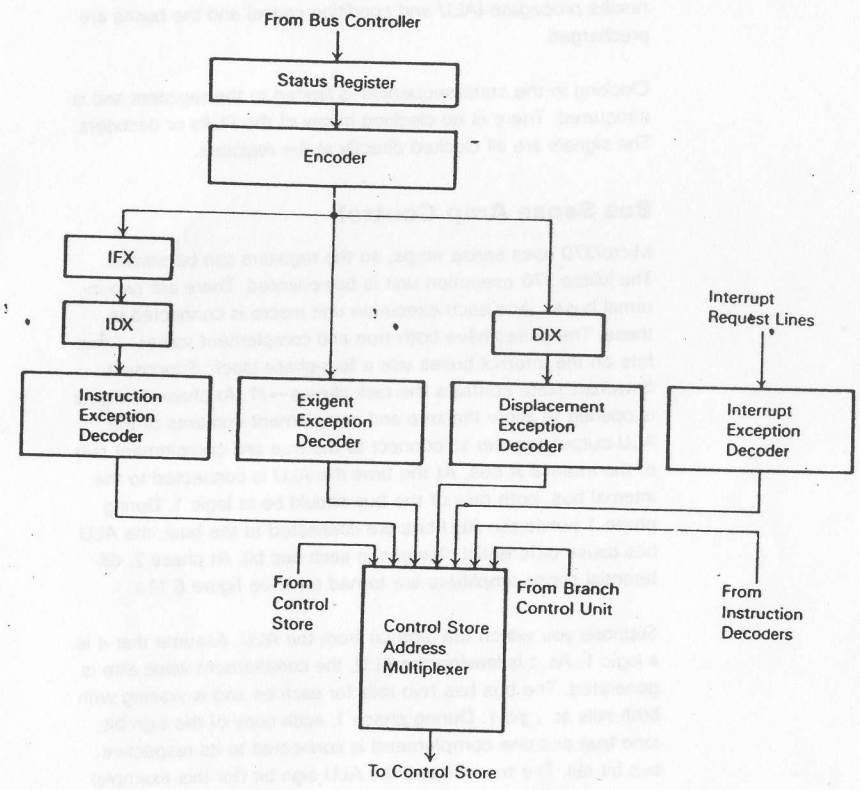


Figure 6.10 Block diagram of prefetch protection registers

multiplexer for the control store next address. Exigent exceptions have the highest priority. Interrupts are second. Instructions are third. Displacement exceptions are fourth. All other control store address sources are mutually exclusive (with each other, not the exceptions) and rank fifth.

Clocking and Timing

Micro/370 uses four-phase clocking. The first phase is the source phase in the execution unit. The operand source gates are opened to the bus. During phase 2, the macros are precharged, and the bus sense amplifiers repeat the source signal on the buses. During phase 3, the destination gate is connected to the bus. The bus signal is strong enough to write over the current contents of the destination register. At phase 4, the results propagate (ALU and condition codes) and the buses are precharged.

Clocking in the state sequencer is limited to the registers and is structured. There is no clocking in any of the PLAs or decoders. The signals are all clocked directly at the registers.

Bus Sense Amp Control

Micro/370 uses sense amps, so the registers can be smaller. The Micro 370 execution unit is bus-oriented. There are two internal buses, and each execution unit macro is connected to these. The buses have both true and complement values. Transfers on the internal buses use a four-phase clock. Suppose a flowchart state contains the task alu→a→r1. At phase 1, a gate is opened to allow the true and complement contents of the ALU output register to connect to the true and complement rails of the internal A bus. At the time the ALU is connected to the internal bus, both rails of the bus should be at logic 1. During phase 1 (while the ALU bits are connected to the bus), the ALU bits cause differential changes in each bus bit. At phase 2, differential sense amplifiers are turned on. (See figure 6.11.)

Suppose you watch the sign bit from the ALU. Assume that it is a logic 1. As it is leaving the ALU, the complement value also is generated. The bus has two rails for each bit and is waiting with both rails at logic 1. During phase 1, each copy of the sign bit (one true and one complement) is connected to its respective bus bit rail. The true rail and the ALU sign bit (for this example)

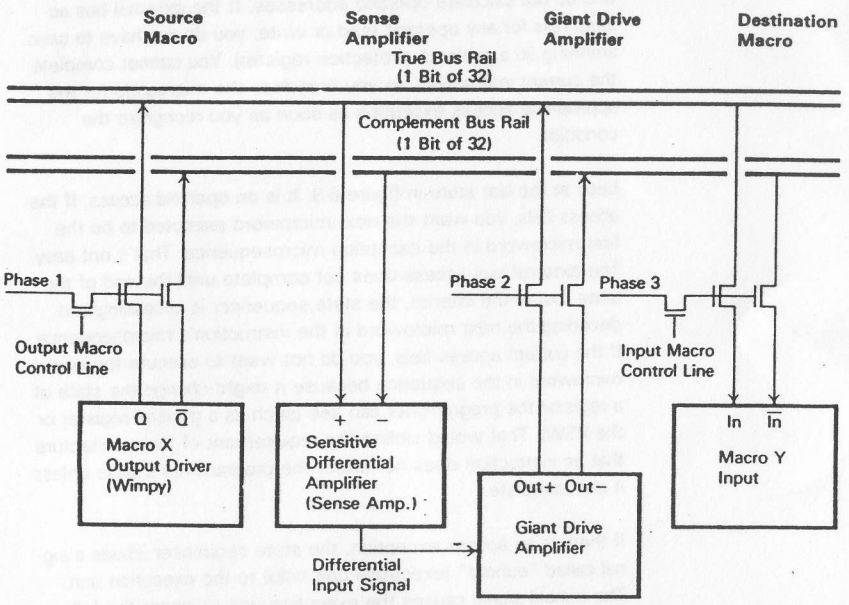


Figure 6.11 Block diagram of 1 bit of bus sense amplifier control

are both at logic 1, so nothing happens. The complement rail is a logic 1 (both of the bus rails always start phase 1 as a logic 1), and the complement of the ALU sign bit is 0. While the gate between the complement bus rail and the ALU output sign bit is open, the complement bus rail value starts to go toward logic 0 (being pulled down by the ALU complement sign bit driver). At the beginning of phase 2, the sense amplifier compares the true and complement rails to see which rail is heading for 0. It then drives the lower rail very fast and very hard toward 0, in this case the ALU complement sign bit. I could not build drivers for each macro capable of driving the entire bus to 0 (or 1) in a short time (because I did not have unlimited space and power for multiple sets of giant drivers). So I built a special differential amplifier that detects what the macro's wimpy driver is trying to do and then uses a giant, fast driver to force the bus to the desired value.

The differential sense amplifiers detect the voltage difference between the true and complement rails for each bit and then

drive the bus in the direction of its tendency. (The tendency is determined by the contents of the ALU bits.) At phase 3, the gate is opened to the register designated by the R1 field of the instruction. The voltages on the A bus simply swamp the R1 register values. This writes the value from the ALU output register into the register designated by R1. At phase 4, the sense amplifiers turn off, and the true and complement rails of the A bus are both driven to logic 1. (This is called precharging.)

The four-phase clocking scheme and sense amplifiers allow the registers to be much smaller because the register output drivers are only large enough to produce a detectable differential voltage on the precharged bus. In effect, all the macros connected to the bus share a common bus driver.

The System/370 registers are all 32 bits. The internal Micro/370 buses also are 32 bits. Some of the System/370 instructions, such as ISK (Insert Storage Key), affect only a single byte of the register. The architecture says the remaining bytes will be unaffected. Other instructions, such as LA (Load Address), affect three bytes of the register and leave the remaining (in this case, high-order) byte untouched. How do you do that with the 32-bit internal buses?

I chose explicit control in the nanoword for the sense amplifier for each bus byte. If you want to transfer only the low-order byte from the ALU output register to the register designated by the R1 field of the instruction, the flowchart task would be alu→a1→r1. The "1" attached to the bus designator tells the sense amplifier control to turn on only the low-order sense amplifier. When the sense amplifier is turned on during phase 2 of the transfer, only the low-order byte of the ALU output register will be repeated on the bus. The other three bytes of the bus will remain at the precharged logic 1 value. When the gate is opened at phase 4, the low-order byte of the R1 register will be overwritten by the value on the A bus. The other three bytes of the R1 register will see only the precharged bus and will not be overwritten. The bus sense amplifiers drive only the low bus-rail, but there is none. The task alu→a7→r1 would cause the three low-order bytes of the ALU output register to be written to the R1 register. If there is no sense amplifier control specified in the task, F (all bytes) is assumed.

System/370 architecture defines two instructions, Insert Characters under Mask (ICM) and Compare Logical Characters under Mask (CLM), which can use more general sense amplifier control. The ICM instruction contains a 4-bit mask. Each bit in the mask corresponds to a byte in a register. The high-order bit corresponds to the high-order byte in the register. The low-order bit corresponds to the low-order byte. The mask can have any value from 0 to 15 (hex F). Contiguous bytes from memory are inserted into the designated register under control of the mask. CLM is similar. It tests contiguous bytes in memory against register bytes corresponding to mask bits. For these instructions, I let the mask field from the IRE control the internal bus sense amplifiers for some states. I form a temporary word with the memory operand bytes in byte positions corresponding to the mask bits. For ICM, I transfer the word to the designated result register with the IRE field controlling the sense amplifiers. The effect is to change only the bytes for which there is a mask value of 1. For CLM, I compare the temporary word with the register word. I mask unused bytes in both words to 0 before I compare them.

Figure 6.12 is a block diagram of the sense amplifier control. In most states, all of the sense amplifiers are running. In some states, the sense amplifiers for the bytes are explicitly controlled by the microcode. In a few states, the microcode gives control of the sense amplifiers to the IRE.

Shifter and Shifter Control

The shifter, like every other element in the Micro/370 execution unit, does only what was demanded by the instruction set implemented. Some System/370 instructions, such as SLDL, shift a register pair, so the Micro/370 shifter is 64 bits. The 64-bit shifter is frequently used as both a 32-bit and a 64-bit shifter. It is used by multiply and divide algorithms for the integer multiply and divide instructions. It is used by Convert to Decimal (CVD), ICM, CLM, and many other instructions.

The microcode uses two basic shift types: explicit shifts and implicit shifts. Explicit shifts tell the shifter exactly what kind of shift and how far to shift. Microword tasks such as "ls1 sh-sl-1" and "rs8 0-sh" tell the shifter explicitly how to shift. "ls1 sh-sl-1" tells the shifter to left shift by one the connected 32-bit sh and the 32-bit sl and to shift a 1 into the vacated low-order posi-

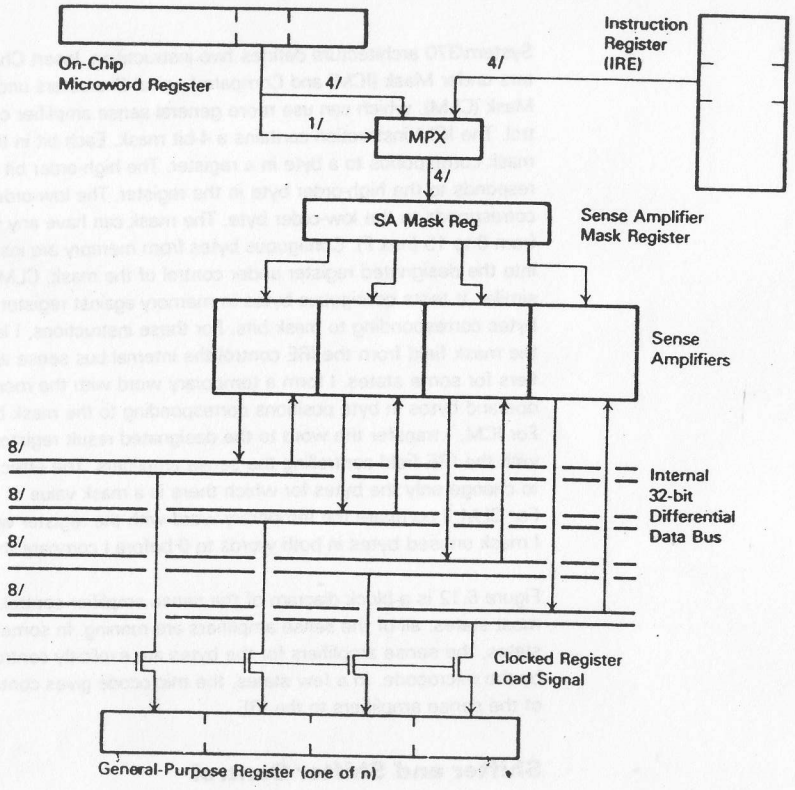


Figure 6.12 Block diagram of bus sense amplifier control by byte

tion. "rs8 0-sh" tells the shifter to right shift 8 bits only the shifter high-order 32 bits and to insert zeros in the vacated positions. Implicit shifts use a microword task of "shift." The shift count is taken from the ALU low-order bit positions, and the direction and shift type are taken from the IRE.

Figure 6.13 is a block diagram of the shifter and the shifter control. Since the execution unit and the internal buses are only 32 bits and the shifter needed 64 bits, it is broken into two (connected) 32-bit pieces. One of the pieces is accessible from each internal bus to allow 64-bit operands to be delivered to the shifter (and shifted) in a single cycle. Figure 6.14 shows the microword

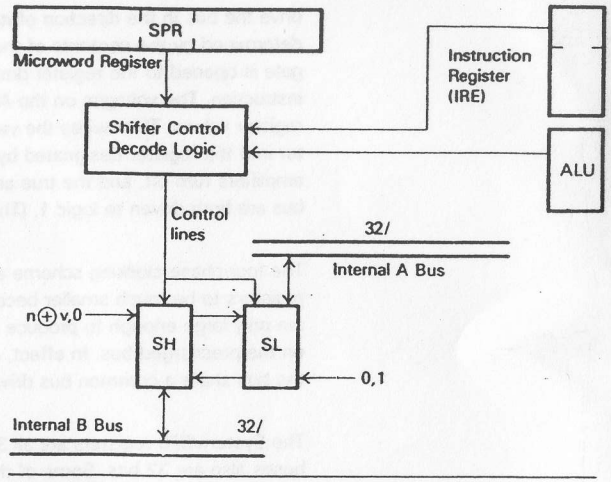


Figure 6.13 Block diagram of shifter and shifter control

SPR 6			
ad→sl	010111	ls8 sh-sl-0;ad→sl	110111
bd→sh	100110	ls16 sh-0	111001
bd→sh1	111000	ad→sl;bd→sh;shift	011111
sh→bd	110001	1xxxxxx→etmssrr	000100
sl→ad	110010	0xxxxxx→etmssrr	001010
sh→bd;sl→ad	110011	x1xxxxx→etmssrr	001100
ad→sl;bd→sh	111111	xxxxx01→etmssrr	000011
rs1 0-sh-sl;ad→sl	100111	xxxxx00→etmssrr	000010
rs1 0-sh-sl;bd→sh	101111	xxxxx11→etmssrr;ad→sl	001111
rs1 n.eor.v-sh-sl;bd→sh	100100	xxx00xx→etmssrr	001001
rs8 0-sh;bd→sh	100101	xxx01xx→etmssrr	001101
rs8 0-sh-sl;bd→sh	101101	xxx10xx→etmssrr	000001
rs16 0-sh-sl	101001	xxx11xx→etmssrr	000101
rs16 0-sh-sl;bd→sh	101100	0011101→etmssrr;ad→sl	000111
rs32 0-sh-sl	101000	0x00011→etmssrr	001011
ls1 sh-0	111011	ad→pur	010000
ls1 sh-sl-0;bd→sh	111100	bd→pur	011101
ls1 sl-sh-0;bd→sh;ad→pur	011110	pur→bd	010101
ls1 sh-sl-1;bd→sh	110101	pur→ad	011011
ls1 sl-sh-0;bd→sh	111110	ad→pur;pur→bd	010100
ls2 sl-sh-0;bd→sh	110110	bd→pur;pur→ad	011001
ls4 sh-sl-0;bd→sh	111101	bd→pur;ad→pur	011100
ls8 sh-0	111010	purh→ad;sh→bd	010011
ls8 sh-0;bd→sh	110100	purl→ad;sh→bd	010001
ls8 sh-0;bd→sh1	110000	none	000000

Figure 6.14 Flowchart tasks and microword control assignments