Chapter 6: Architecture and Design of System Controllers, and the Design of the OM2 Controller Chip

Sections:

This chapter presents alternative structures for controlling a data path of the type described in chapter 5. It contains a review of the basic concepts of the stored program computer, and how such computers are constructed from a combination of (i) a data processing path, (ii) a controller, and (iii) a memory to hold programs and data. A description is given of some of the ideas behind the architecture of a specific controller chip, designed at Caltech, for use with the OM2 data chip. Several examples of controller operations are provided.

We have previously used the OM2 data path chip as a source of illustrative examples, primarily at the *circuit layout* level, to help the reader span the range of concepts from devices, to circuit layout, to LSI subsystems. In this chapter, the controller chip is used as a source of examples one level higher, at the *subsystem* level, to help the reader span the range from digital logic circuits, to LSI subsystems, to arrangements of subsystems for constructing LSI computer systems. The computer system one can construct using the OM2 data chip, the OM2 controller chip, and some memory chips, contains rather simple, regular layout structures. Yet the system is functionally quite powerful, comparing well with other classical, general-purpose, stored program computers.

All present general-purpose computers are designed starting with the stored program, sequential instruction fetch-execute concepts described in this chapter. These concepts are important not only for understanding present machines, but also for understanding their limitations.

As we look into the future and anticipate the dimensional scaling of the technology, we must recognize that it will ultimately be possible to place very large numbers of simple machines on a single chip. When mapped onto silicon, classical stored program machines make heavy use of a scarce resource: communication bandwidth. They make little use of the most plentiful resource: multiple, concurrent, local processing elements. What might be the alternatives? We will reflect on some of these issues at the end of this chapter, and examine them in detail in chapter 8.

Alternative Control Structures

In this section we will clarify the distinction between the data processing and control functions in a digital computer system, and then examine several alternative forms of control structures.

The data processing path described in chapter 5 is capable of performing a rich set of operations on a stream of data supplied from its internal registers or from its input/output ports. How is it that a structure having such a static and regular appearance as the OM Data Path can mechanize such a rich set of operations? An analogy may help in visualizing the data path in operation. Imagine the data path as like a piano, with the interior regions of the chip visualized as the array of piano wires, and the control inputs along the edge of the chip as the keys. Under the external control of the controller chip, now visualized as the piano player, a sequence of keys are struck. During some cycles, many keys are struck together simultaneously, forming a chord. A complex function may thus be performed over a period of time by the data path, just as the static-appearing array of piano wires may produce a complex and abstract piece of music when a series of notes and chords are struck in a particular order.

We see from this analogy, however, that the data path in itself is not a complete system. A mechanism is required to supply, during each machine cycle, the control bits which determine the function of the path during that cycle. The over: ll operations performed on data within the data path are determined by sequences of control bit patterns supplied by the system controller.

Mechanisms for supplying these sequences of control inputs to a data path can either be very simple or highly complex. There are many alternative sorts of control structures. The detailed nature of the controller has many important effects on the structure, programming, and performance of the computer system. Let us begin with the description of the simplest form of finite state machine controller. Then, through a sequence of augmentations of this controller, we will build up to the concepts of the stored program computer and microprogramming.

Simple block diagrams, such as figures 1, 2, and 3, are used in this chapter to convey the essential distinctions between various classes of controllers without requiring the diagramming of the internal details of any particular controller. Although the detailed internal logic of any particular controller may be rather complex, there are only a small set of key ideas involved in the hierarchy of controller structures presented by the sequence of block diagrams.

If you closely examine the controllers of typical computers, you will find that every one either is,

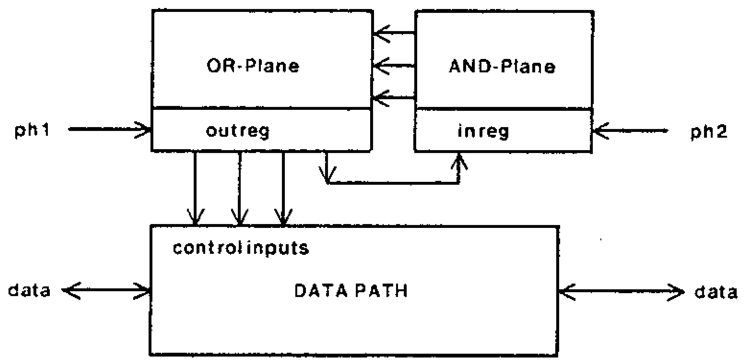


Fig.1. Finite state machine controlling the Data Path

In this case, periodically cycling thru a fixed sequence of states

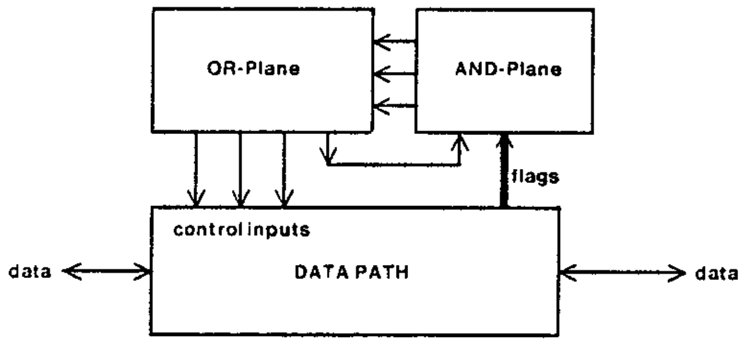


Fig.2. Finite state machine controlling the Data Path

In this case, the next state can be a function of the previous operation's outcome

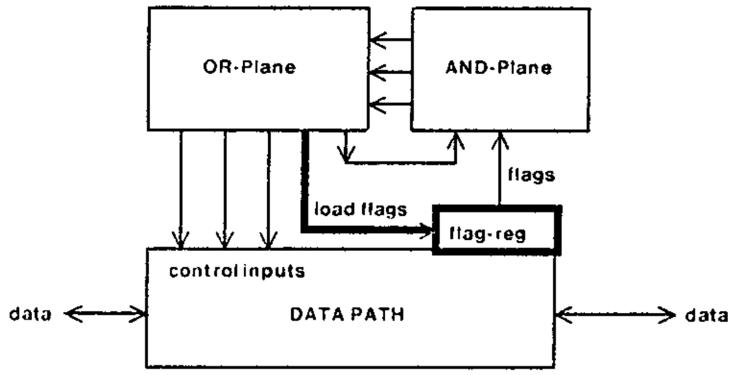


Fig.3. Finite state machine controlling the Data Path

In this case, a data path operation result may control machine sequencing for a number of later cycles

or contains within it, a finite state machine such as those described in Chapter 3. The very simplest form of controller for the data path is a finite state machine having no inputs other than state feedback lines, as shown in figure 1. The operations performed by the data path are determined by the sequencing of the state machine. Each clock cycle, the output of the OR plane is fed back into the AND plane and determines the next state of the state machine, which periodically cycles through a fixed sequence of states. The data path is clocked in synchronism with the controller, although for simplicity we haven't shown clock inputs to the data path in the figures. Thus a fixed algorithm implemented in the code of the state machine operates on the data in the data path.

Such a control structure could be used with the data path to implement a function such as a digital filter, in which data is taken in from the left port of the data path, a fixed set of operations performed on the data, and a result output at the right port of the data path. However, this elementary control structure provides no way to perform operations which depend on the outcome of a previous operation or upon the data itself.

A simple augmentation, shown in figure 2, enables the control sequencing to be a function of the outcome of the previous operation. In figure 2, some of the data, or some logical functions of the data, called *flags*, are fed into the AND plane inputs of the state machine along with the next state information. Some typical flags are: whether or not the ALU output is zero, is positive, or whether or not one ALU input is numerically equal to the other. The next state can thus be a function of flags generated during the preceding operation. To simplify figure 2, we have not shown the clock inputs to the PLA. However, assume that all subsystem structures shown in the figure, and throughout this chapter, are appropriately operated in a synchronous manner using our normal two phase clock scheme and proper design methodology.

While in principle the figure 2 structure is quite general, improvements are possible which allow greater flexibility and compactness of representation of the algorithm in the state machine. One of these improvements is shown in figure 3. Here an additional output from the OR plane of the state machine is used to control the loading of the flag outputs of the data path into a flag register. The flag register is used as an input into the AND plane of the state machine. This enables flags generated by a particular operation to be used as control inputs for the state machine for a number of later operations. The stored flag values are replaced by a new set only when the flag load signal is raised. One difficulty inherent in this structure is the limited amount of information provided by the few flags generated by the data path's ALU.

The Stored Program Machine

A very general and powerful arrangement is shown in figure 4a. This structure is similar to the one discussed in the last section. In this case the state machine sequencing is controlled not only by the last state and flags, but also by the data coming from some memory attached to the machine. The memory contains not only the data upon which the data path is operating, but also contains encoded information for influencing the sequencing of the state machine.

This scheme gets around the limitation of the structure in figure 3, and also provides a complete new dimension of possibilities. The basic idea is to design the state machine controller so that it may perform *any of a set* of different predefined operations, called the *machine instruction set*, rather than just perform one dedicated, predefined operation. This machine instruction set is carefully defined so as to enable the system composed of the data path, controller, and memory to mechanize any of a number of different algorithms of interest to a number of different users. These algorithms are implemented as *programs* composed as *sequences of machine instructions* loaded into the memory. These programs operate upon data also contained in the memory.

It is possible to show that this arrangement is perfectly general and can implement any digital data processing function. John von Neumann[1] is generally credited with originating this idea of a stored program machine, and such machines are often called von Neumann machines. The abstract notion of the most basic form of stored program machine was pr· posed by Turing[2] in 1936, for application in the development of the theory of algorithms. The abstract *Turing Machine* is important not only for historical reasons, but also because of its present use in the development of the theory of computational complexity of sequential algorithms.

The way in which the stored program machine operates is as follows. One of the internal registers of the data path is selected to hold a pointer into the program stored in the memory. This register is commonly called the *program counter* (PC), or alternatively, the *instruction address register*. In one particular state of the controlling state machine, which we will call the *fetch next instruction* (FNI) state, the program counter is caused by the state machine to output its data as an address to the memory, and the state machine initiates a memory read from this address. The data from this memory read operation is taken into the AND plane of the state machine, placing the state machine into a state which is the first of the sequence of states which mechanize the machine instruction corresponding to the code just read from the memory. The state machine then sequences the data path through a number of specific operations sufficient to perform the

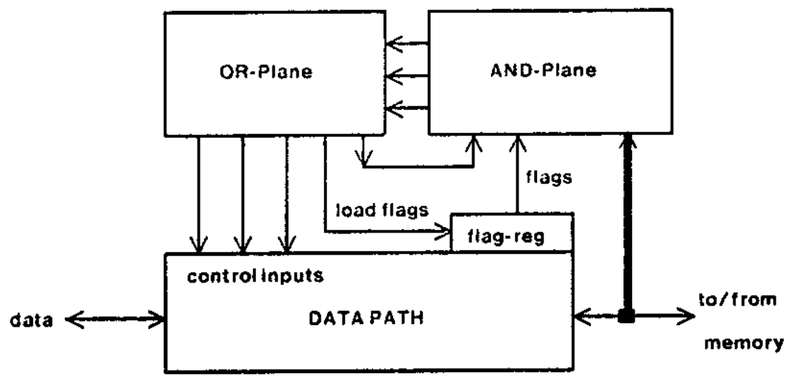


Fig.4a. A Simple Stored Program Machine
Where data read from a memory
can affect machine sequencing

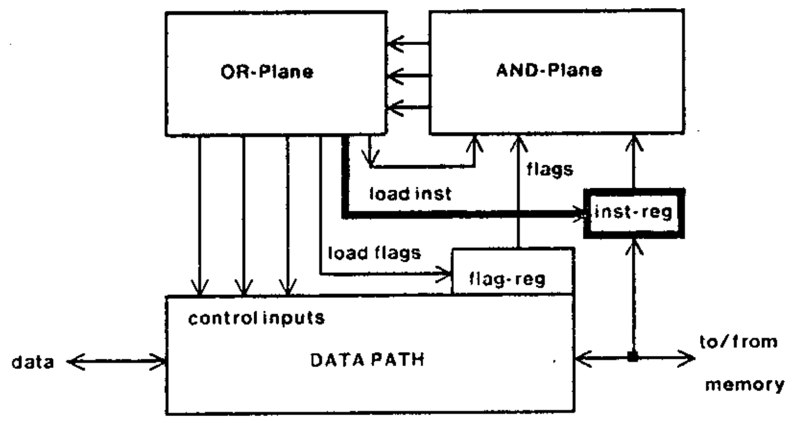


Fig.4b. A Simple Stored Program Machine
In this case, augmented by an instruction register

function defined by that instruction. At some point during instruction execution the next PC value is calculated, usually by simply incrementing the current PC value.

When the state machine has completed the interpretation, or execution, of the machine instruction, it returns to the FNI state. The instruction fetch is then repeated, sending a new program counter value to the memory as an address, reading the next instruction from the memory, and beginning its interpretation. The system can thus perform any set of required operations on data stored in memory, as specified by encoded instructions stored in memory.

There is a problem with the organization of the controller in figure 4a. Most of the steps of an instruction execution sequence need as input the encoding of the instruction which initiated the sequence. In figure 4a, this information must be duplicated each cycle by the next state information. The number of bits in the feedback path for this information can be reduced by the arrangement shown in figure 4b. Here the incoming instruction is stored in a register, called the *instruction register* (IR), which is loaded under the control of an output from the state machine. It stays in the instruction register, and is available for state machine input during the entire period that particular instruction is being interpreted by the machine. This new arrangement is not fundamentally different from the preceding one, but is more efficient in its use of the PLAs.

The separation and naming of the instructi on register also enables us to take another step in the structuring of the state machine controller's operations: the conception and naming of stages of the interpretation of instructions fetched and held in the IR.

Suppose we have defined a machine instruction set which, for example, includes arithmetic-logic instructions, memory instructions, and branch instructions. Suppose we also have a data path such as the OM data chip, or any other typical data path containing registers, an ALU, buses for moving data around, and inputs for control signals to control the movement of data and the ALU operations. What functions must a control unit, such as that shown in figure 4b, perform in order to fetch and execute machine instructions? We find that in most stored program machines, the execution or interpretation of each machine instruction is typically broken down into the following *six basic stages.* Note that some instruction types may skip one or more of the stages, and that each of the stages may require sequencing through several controller states:

(1) *Fetch next instruction:* This is the starting point of the fetch-execute sequence. The machine instruction at the address contained in the PC is fetched from the memory into the IR.

(2) *Decode Instruction:* As a function of the fetched machine instruction's type, encoded in its OP code field, the controller must "branch" to the proper next control state to begin execution of the operations specific to that particular instruction type.

(3) *Fetch instruction operands:* Instructions may specify operands such as the contents of registers or of memory locations. During this execution stage, the controller cycles through a sequence of states outputting control sequences to fetch the specified operands into specified locations, for example into the input registers of the ALU.

(4) *Perform Operation:* The operation specified by the OP code is performed upon the operands.

(5) *Store Result(s):* The results of the operation are stored in destinations, such as in registers, memory locations, flags, etc..

(6) *Set up next address, and return to FNI:* Most instructions increment the PC by one and return to the FNI state (1). Branch instructions may modify the PC, perhaps as a function of flags, by replacing its contents with a literal value, fetched value, or computed value.

Now, how would we go about designing such a controller? We can construct the state diagram for the controller just as we did for the traffic light controller example in chapter 3. Then we proceed to build up the detailed state transition table, and finally derive the AND and OR plane code for the PLA. However, in this case the state diagram will be rather more complex than that in our earlier example. One hundred or more states may be required to implement the controller for a simple machine instruction set. How do we even begin constructing the state diagram? The above list of stages of instruction execution provides a simple means of structuring the diagram. Figure 5 contains part of the controller state diagram for a typical stored program machine. The diagram is structured as a matrix of regions, where the instruction execution stages proceed from top to bottom, and the columns contain specific state sequences for each instruction type. The FNI state is placed at the top of the diagram, followed by the states leading to the decode. The decode results in a many-way branch, each path leading to a sequence for executing a particular instruction type. The figure contains some (informal) details indicating the sorts of specific control operations performed at each stage of the instruction execution or interpretation. One will encounter many variations on the simple state diagram structure shown. These are usually easily understood elaborations. For example, groups of machine instructions may share common subsequences of control operations. To reduce the number of states, we might have another level of decoding, first decoding to groups of instructions and performing shared operations, then

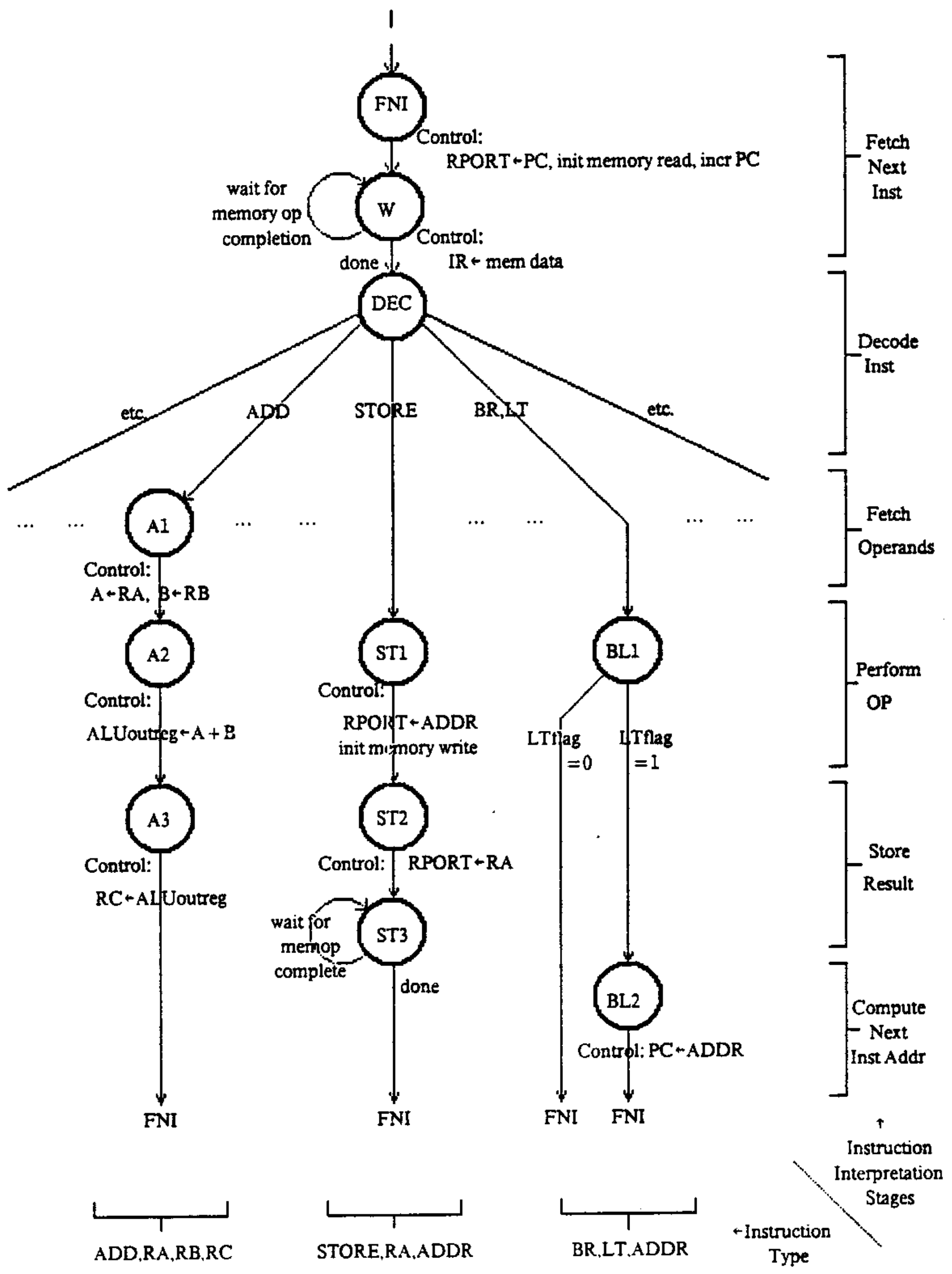


Fig.5. A Portion of the State Diagram for the Controller of a Stored Program Machine
[Illustrating some typical instruction interpretation state sequences, and their associated control outputs]

(dscstdiag.press)

decoding to individual instruction types. In any event, the generation of the state diagram and eventually the PLA code is just a matter of grinding out the details. The generation of these details is another activity which is made more tractable by following a structured approach.

Some examples follow which will clarify how a machine instruction's execution can be divided into parts, and how the parts interact with each other. Instead of using the graphical notation of figure 5, an informal tabular form is used, containing a list of statements that are normally performed sequentially, as encountered. In these examples, the unbracketed statements under "control [& state] sequence" indicate control actions. However, some statements explicitly set the next state Y'. These statements are bracketed, "[ ]", and indicate a more complex state transition than simple state-to-state progression (shown in figure 5 by a single arrow between circles).

ALU Example: Suppose that an arithmetic/logic instruction in our machine instruction set has the general form: { ALUOP, REGA, REGB, REGC }, specifying that ALUOP be performed on operands REGA and REGB, and the result stored in REGC. Then the instruction { ADD, R7, R2, R5 } might be executed by the following control sequence. Note that certain of the individual control steps may occur in the same machine cycle (for example: A←R7, B←R2), as a function of the capabilities of the data path; the more the data path can do in parallel, the fewer machine cycles it will require to complete an instruction:

Function of sub-sequence:	*Control [& State] sequence:*	*Comments:*
Fetch Next Inst:	RPORT ← PC	Place next instr. address in right port.
	read memory	Raise control line to initiate memory read.
	PC ← PC+1	Increment PC, overlapping incr. with fetch.
	[Y' = fcn(memop complete)]	Loop here till memory read completes.
	IR ← mem data	Load IR with inst. when read completed.
Decode Instruction:	[Y' = fcn(IR)]	Set machine state as fcn of instruction.
Fetch Operands:	A ← R7	Load ALU input registers with operands.
	B ← R2	
Perform Operation:	ALUoutreg ← A+B	Add A and B, store in output register.
Store Result:	R5 ← ALUoutreg	Send result address to R5.
	[Y' = FNI]	Inst. not a branch, so simply return to FNI state

The example assumes there is some sort of shared access to the memory, and thus the time for completion of memory accesses is not predictable. That is why we wait, testing for the presence of a completion signal before proceeding. In some computer systems, such memory accesses might proceed in lockstep with the controller sequencing, and the data taken from, or placed on, the memory bus at some fixed number of cycles following initiation of the memory operation.

Normally, most machine instructions are not branches, so we usually just have to increment the PC sometime during instruction execution. This incrementing can often be overlapped with other control operations. In the example, the incrementing of the PC is done during the FNI stage, while waiting for the completion of the instruction-fetch memory operation.

Memory Example: A memory instruction in our set might have the general form { MEMOP, REGA, ADDRESS } , specifying the loading or storing, according to MEMOP, of the contents of register REGA to or from the memory address ADDRESS. The instruction { STORE, R3, ADDRESS } might then be executed by the following control sequence:

Function of sub-sequence:	*Control [& State] sequence:*	*Comments:*
Fetch Next Inst:	RPORT ← PC	Place next instr. address in right port.
	read memory	Raise control line to initiate memory read.
	PC ← PC+1	Increment PC.
	[Y' = fcn(memop complete)]	Loop here till memory read completes.
	IR ← mem data	Load IR with inst. when read completed.
Decode Instruction:	[Y' = fcn(IR)]	Set machine state as fcn of instruction.
Perform Operation:	RPORT ← IR(ADDRESS)	Send the address of the result to the memory.
	write memory	Raise write control line to init. memory write.
Store Result:	RPORT ← R3	Place result in right output port.
	[Y' = fcn(memop complete)]	Loop here till memory write completes.
	[Y' = FNI]	Inst. not a branch, so simply return to FNI state

Branch Example: Suppose that branch instructions have the form: { BR, COND, ADDRESS }, specifying that if the condition COND is true according to the flags, then the PC is to be loaded from memory address ADDRESS. The branch instruction { BR, LT, ADDRESS } might then be executed by the following control sequence:

Function of sub-sequence:	*Control [& State] sequence:*	*Comments:*
Fetch Next Inst:	RPORT ← PC	Place next instr. address in right port.
	read memory	Raise control line to initiate memory read.
	PC ← PC+1	Increment PC.
	[Y' = fcn(memop complete)]	Loop here till memory read completes.
	IR ← mem data	Load IR with inst. when read completed.
Decode Instruction:	[Y' = fcn(IR)]	Set machine state as fcn of instruction.
Perform Operation:	[Y' = fcn(LTflag)]	Set machine state as fcn ALU LTflag. Set to FNI if notLT. Else continue and generate new address.
Next Address:	PC ← IR(ADDRESS)	Generate new next address.
	[Y' = FNI]	Return to the FNI state.

Now, how are the next higher level system software control functions mapped onto this basic machine structure? Higher-level functions common to all machine instructions are often performed within the FNI stage of instruction execution. After return to the FNI state, but prior to the decode state, one machine instruction has been completely executed but no action has yet been taken to execute the next instruction. Therefore, that is a natural place to check for interrupts from I/O devices, to test the priorities for task switching in a multiprogramming environment, and so forth. The testing of these logical signals, which are input to the state machine, can often be overlapped with other FNI activity. Multiple tasks may then be implemented by having the controller manipulate multiple program-counters.

In summary, once both a machine instruction set and a data path have been defined, then the control sequences required to interpret the machine instructions can be "programmed", the overall controller state diagram constructed, the "code" for the AND and OR sections of the state machine can be generated, and software systems can be built upon the resulting stored program machine. Interestingly, the control sequences in the above examples look somewhat like "programs" written in a very primitive machine language. This observation anticipates the concept of microprogrammed control, which is described in the next section.

For more information on this material, including the various trade-offs involved in the definition and encoding of instructions, see the many examples in Bell and Newell[6]. See also Dietmeyer[7], which works out an example all the way from state diagram through the design of the controls of an elementary digital computer. Formal methods for describing state machine algorithms are given in reference 7, and in the reference R4 of chapter 3; an interesting alternative method based on ideas of T. E. Osborne, is presented along with practical examples in Clare[8].

The abstract concepts behind the arrangement shown in figure 4b are used in almost all stored program digital computers manufactured today. A computer having any sort of machine instruction set can be implemented with the arrangement shown. In many cases, the state machine is implemented in random logic and therefore is not easily recognizable as one of the forms shown. However, the operations performed are equivalent to those described here. Note that any performance contraints imposed by limited functionality in the data path simply trade off against the number of machine cycles required to mechanize particular algorithms.

Microprogrammed Control

Sometimes the complete machine instruction set is not definable at the time a computer is being designed. This contingency often arises when certain operations, defined by some later user, must be executed at very high speed. Perhaps the data path is inherently capable of satisfying the required performance constraints, but not when operated under the control of any sequence composed of standard machine instructions. In such cases, special new machine instructions would have to be defined and then implemented in the state machine control logic.

Another common situation is the need to execute the instruction set of another computer system for which the user has existing programs. While such instructions could be executed by simulation, i.e. by interpreting them via a program written in the original machine instruction set, such simulations usually pay a high performance penalty. It would be much better if the machine could execute them directly. However, a substantial augmentation and/or modification of the controller's logic would have to be made, for such direct execution to be possible.

In both of these situations it would be desirable if the state machine were implemented in some writeable medium, rather than in the fixed code of a standard programmable logic array and thus patterned permanently in the silicon. While it is quite possible to build writeable programmable logic arrays, none are currently in use. Instead, machine designers have invented many clever ways of using standard writeable memories to hold the feedback logic of the state machine.

The simplest such arrangement is shown in figure 6. Here the state machine is implemented using a set of memory chips. Collectively, this set of memory chips functions externally exactly as the programmable logic array shown earlier. However, this very elementary structure has a problem in supporting wide machine instruction words, since the decoder must exhaustively decode all combinations of the input variables. Thus, if *f* is the number of flag bits, and *n* is the number of next state lines, then the memory must have $2^{(i+f+n)}$ words to be of sufficient size to allow emulation of any machine having instructions *i* bits wide. For this reason designers have taken to inserting more complex logic than just a simple instruction register into the path between the data source and the memory decoder section of state machines of this form.

A system using a logic path between the memory bus, or source of instructions, and the memory decoder section of the state machine is shown in Figure 7. Here a logic block we have termed the *micro program-counter path* is inserted between the source of machine instructions and the

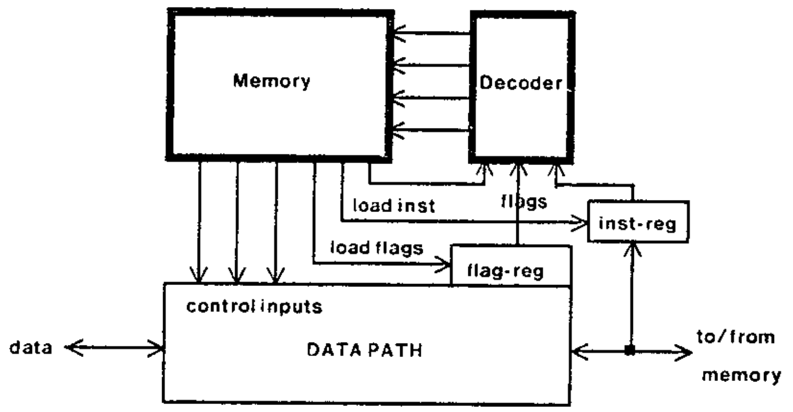


Fig.6. An alternative form of Stored Program Machine
Illustrating the use of a Decoder and Memory to implement
the state machine controlling the data path

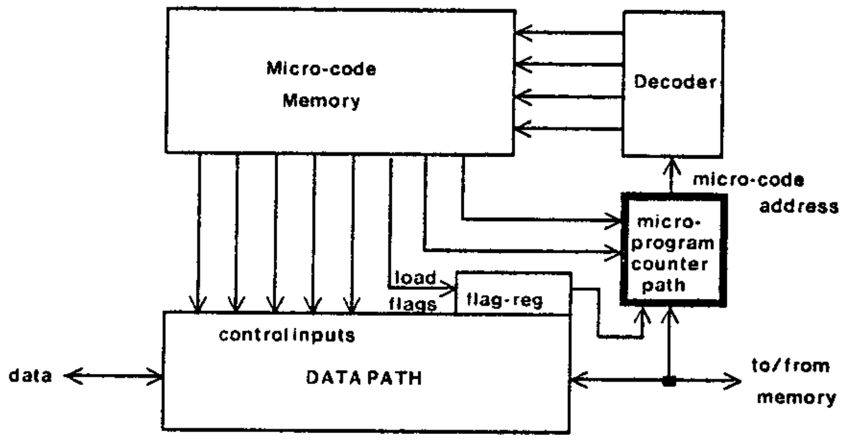


Fig.7. An alternative form of Stored Program Machine

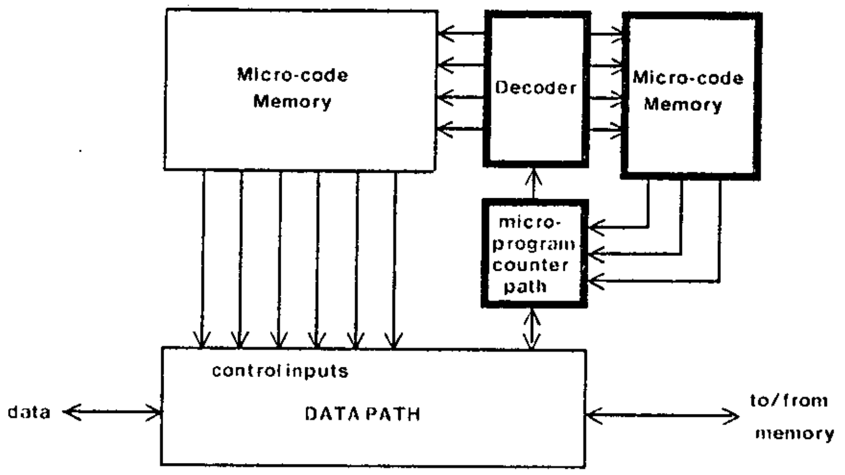


Fig.8. Another Way of Visualizing the Figure 7 Machine

inputs to the decoder. This type of control, using either writeable or read-only memories, is generally referred to as *microprogrammed control.* Notice in figure 7 that the flags and the machine instruction fetched from the memory both act as input data to the small micro program-counter data path, and the outputs of this data path are the microcode memory address lines. The arrangements shown is very powerful and general, and capable of emulating any instructions set for which there is sufficient microcode memory.

In a microprogrammed controller, the design of the control logic is reduced to encoding sequences of control bit patterns to be stored, along with control memory address sequencing information, in the microcode memory. The encoded control bit patterns for each clock cycle or machine cycle are visualized, as in the examples in the past section, as a primitive form of "instruction" and are called microinstructions. Rather than creating a "circles and arrows" state diagram and "assembling" PLA code, we write a symbolic microprogram and assemble it in the same manner as we would a symbolic machine language program.

The micro program-counter data path (μPC) is similar to the main data path: it is controlled by a number of outputs from the microcode memory section of the state machine. Its main purpose is to decrease the amount of microcode memory required to emulate the particular machine instruction set being implemented. This is done in two ways: First, the μPC maps the f+n bits of state into a smaller number of bits which are then decoded to address the microcode memory. Secondly, it reduces n by allowing complex operations within the μPC to be specified with only a few bits of control information. The controller chip described in the later sections of this chapter is the microprogram counter path portion of a microprogrammed controller for OM2.

The concept of microprogramming was originated by M. V. Wilkes[3,4] in 1951. In those days when controller logic functions were implemented using gates constructed out of vacuum tubes, switching hardware was very expensive compared to wires, and great efforts were expended towards gate minimization. This inevitably led to rather intertwined connections in the controller logic, and any change in function might require a complete redesign. Wilkes presented the notion of microprogrammed control using a read-only memory to hold the control sequences, as a means of bringing regularity and structure to the design of system controllers and thus simplifying their design and redesign. There is a large body of knowledge associated with the architectural implications of microprogrammed control, and the serious reader will benefit from a study of the literature[5,6,7].

Today, although we can easily implement control logic in a structured way using a PLA, we still often use microprogrammed control in order to obtain the advantages offered by writeable control logic. An additional present advantage of microprogrammed control is that the detailed design/redesign of control logic is extended into the wide arena of those familiar with linear sequential programming concepts. In the future as the "programming" of structures into silicon becomes easier, as the time to implement designs becomes much shorter, and as state machine "coding" becomes more widely understood, we may find that these activities will become viewed as a natural extension of microprogramming.

There is an alternative way of viewing the machine shown in figure 7. Examine carefully the loop formed by the micro program-counter data path, the decoder section of the microcode memory, and the outputs of the microcode memory which are used to control the micro program-counter. We can view the microcode memory address as an instruction address and the wires coming from the microcode memory to control the micro program-counter path as an instruction.

This alternative view is illustrated in figure 8. Observe that we have constructed another stored programmed machine of the same form as that shown in figure 4b. We have come full circle in our machine design: in our zeal to put as much capability as possible in the path between the machine instruction and the decoder of the state machine, we have in fact created a stored programmed machine within a stored programmed machine. This phenomenon is referred to by Ivan Sutherland as the "great wheei of reincarnation". Computers often have many such levels of machine within them, each a general purpose stored program machine in its own right. We thus find that elaborate computing machines are often only simple machines, nested and connected together in complex ways.

Design of the OM2 Controller Chip

We now describe some of the ideas behind the design of one particular micro program-counter path used for controlling the OM data chip in the system configuration[9] described in chapter 5. The design of the controller chip will be examined at several stages in its actual development. This material illustrates the mapping into LSI, and the topological/geometrical planning in LSI of various subsystems such as stacks, incrementers/decrementers, multiplexers, etc., which are useful in constructing controllers.

Even at the 1978 value of $\lambda = 3$ microns, the OM2 data path and certain forms of controller could be integrated onto a single chip. The separation of these modules onto two chips was primarily for research and tutorial purposes in the university environment: so that different controllers could be used with the OM2 data chip and vice-versa. The fact that data path and controller are on separate chips does, however, lead to detailed system partitioning decisions aimed at minimizing interchip communication. These decisions might be made differently were data path and controller integrated onto the same chip. Nevertheless the issue of minimization of interchip communication would still be involved at the next system level, and is worthy of study.

The basic function of the *micro program-counter path*, which we call *the controller* for short, is to provide microprogram memory addresses. The microprogram memory addresses are stored in a latch which is called the *micro program-counter*, or μPC. The μPC should be distinguished from the *program-counter*, or PC, which stores the main memory addresses of higher level machine instructions. The most common address calculation is to increment the address by one, so in addition to the μPC latch, the controller should contain an incrementer. The second most important address calculation is the jump or branch, so there should be some means of forcing values into the μPC latch. With the hardware mentioned so far, we have progressed one step beyond the controller type shown in figure 6: our instruction register also increments, so we don't need the feedback terms that originate in the microcode memory and drive the memory decoder.

A great deal of microcode memory space can be saved if *subroutines* are available at the microcode level. These subroutines can be shared between microcode sequences emulating instructions at a higher level. For example, many different machine instruction types may have the same set of operand fetch sequences. If the machine instruction set encodes a variety of indexing or relative addressing schemes, these operand fetch sequences may be quite lengthy, and repeating these sequences for every instruction type would waste a great deal of microcode memory. To provide such microcode subroutine capabilities, provisions must be made for saving μPC values, which is most easily done with a stack. Stacks are easily constructed in LSI. An example of stack cell and subsystem design, and stack control driver design, is given in chapter 3.

The microcoder may also wish to use *relative* jumps or subroutine calls so that relocatable microcode can be written. To provide for relative operations, an adder must be included that can add displacements to the μPC contents. The displacements can either be fixed displacements and come from the microcode or be calculated displacements and come from the data path. Calculated displacements enable many-way branching, or *dispatching*, in the microcode, which is an almost essential operation for emulating instructions at a higher level. An example of dispatching will be given in a later section. Therefore, provisions should be made for accepting displacements from either the microcode or the data path.

Another microcode address operation that could be considered is a form of loop operation, which is useful when sections of microcode should be executed *n* times, where *n* can either be a constant and come from the microcode or be the result of a calculation done in the data path. One way to implement this instruction is to dedicate one register in the data path to be the loop counter and to do conditional branches in the controller based on the result of decrementing the value in that register. This is simple to do, because the hardware of the controller and data path discussed so far will allow the execution of this instruction. Unfortunately, there is a time penalty when doing interchip communication: the loop counter must be decremented during one cycle, the result of the decrement must be sent to the controller during the following cycle, and a conditional branch must be performed in the controller on the third cycle. If the loop counter were in the controller chip, this operation would only take one cycle and would not require the use of the ALU for one cycle in the data path.

With only one loop counter, loops could not be nested, and loops could not be used inside of subroutines. If a stack were provided for the counter values, however, nested loops and loops within subroutines could both be accommodated.

The first OM2 controller proposal was based primarily on the arguments presented above. Figure 9 shows a block diagram of the proposed controller. Table 1 lists the operations possible for each of the three sections of the controller chip: the μPC source selection, the μPC stack operation, and the loop stack operation. In each cycle, the controller executes one operation in *each* of the three sections. For most operations, all three sections work together to perform the programmed

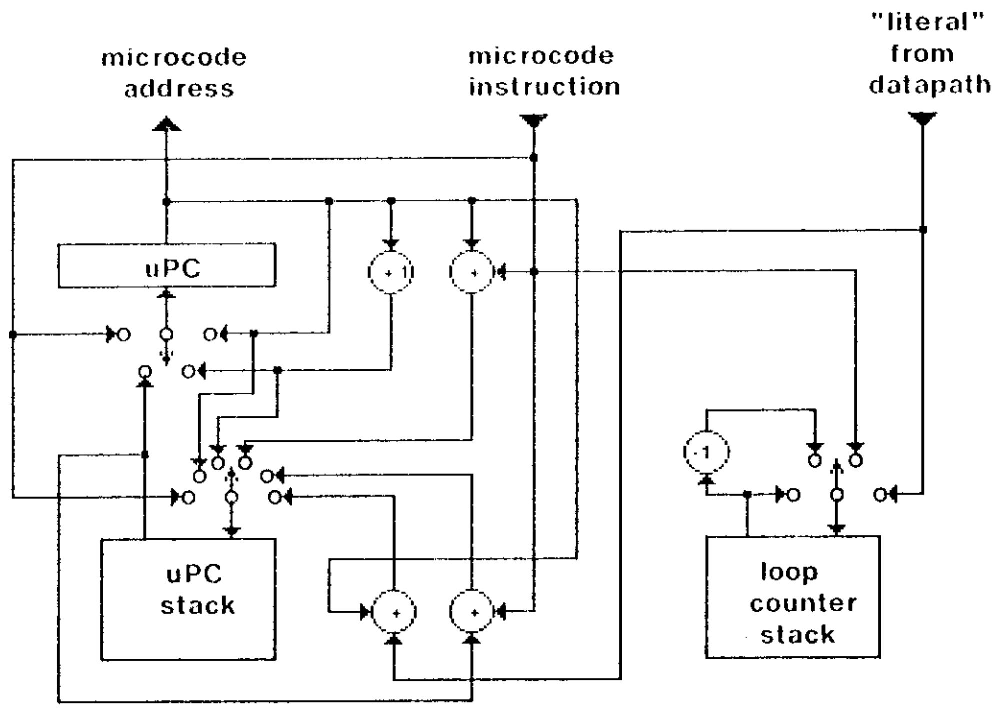


Figure 9. Initial Block Diagram of the Controller Chip.

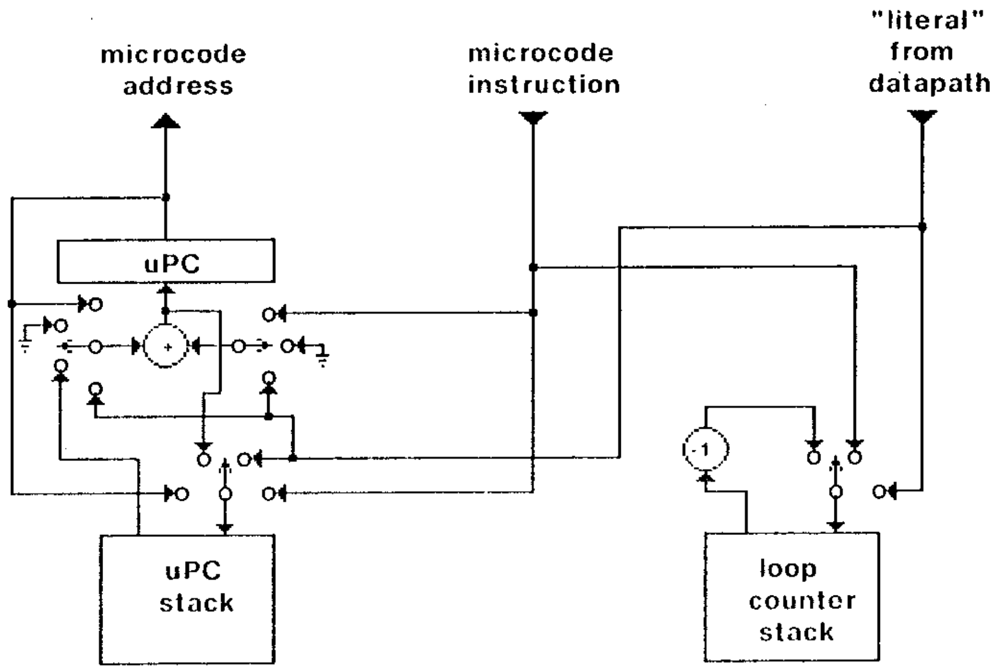


Figure 10. Final Block Diagram of the Controller Chip.

uPC Sources:

uPC + 1
microcode
uPC Stack Top
True: uPC Stack Top; False: uPC + 1

Loop Stack Operation:

Push microcode
Push literal
Push count
Pop
Decrement Count
NOP

uPC Stack Operation:

Push uPC + 1
Push microcode
Push uPC + microcode
Push uPC Stack Top + microcode
Pop
Push uPC + literal
True: Pop; False: NOP
True: NOP; False: Pop
NOP

Table 1. Opcodes of the Initial Controller Proposal.

uPC Sources:

uPC + 1
uPC + microcode + 1
microcode
Stacktop + 1
Stacktop + microcode + 1
Stacktop + literal + 1
uPC + literal + 1
literal + microcode

Condition Selection:

False
True
Data path flag
Compliment of Data path flag
Count = 0
Count < > 0
Data path flag AND Count = 0
Data path flag OR Count = 0

uPC Stack Sources:

Adder output
uPC
microcode
literal

Counter Operations:

No Operation
Push microcode
Push literal
Pop to literal bus
true: decrement; false: pop
true: decrement; false: nop

uPC Stack Operations:

Push if condition is true
No push
Pop if condition is true
No pop

Table 2. Opcodes of the Final Controller Design.

operation. There are cases, however, when only one or two of the sections are needed to perform the controller's instruction, so the other section(s) are free to perform other tasks. For example, the loop stack may be loading a count from the Data Path, while the μPC sections are performing a subroutine call. This concurrency saves having to load the count later, and may save microcode space. Because the controller's instruction is broken into three fields, more than one thing can be happening in parallel in the controller. This is why the instruction was not kept as one field and decoded into the three sections on chip.

The controller shown could handle all of the microcode address operations listed above, and a few new operations were discovered and added to the list. However, there are a few problems with this design. It is a "brute force" design: rather than viewing the whole chip at one time and looking for generalizations, each section of the chip and of the chip's operation was looked at individually and the chip was filled with specialized hardware for performing specialized operations. It was found that by adding one circuit here a new operation could be performed, and that by adding another circuit there a different operation could be added to the repertoire. Many designs suffer from "creeping features" of this sort. While it may be easy to draw circles and arrows on paper, it can be more difficult to draw adders and multiplexers on silicon. It would be very difficult to route all the wires needed to interconnect the devices shown in the proposal.

So let's make a few generalizations about the circuits in the design. First, there are too many adders on the chip. A close look a the proposal shows that for almost all operations we only use one adder for any one cycle, and the few operations that used more than one adder are not critical operations. Incrementing the μPC can also be done in the adder, by clearing one of the data inputs to the adder and forcing a carry into the first stage. Thus, all three of the adders and the incrementer can be combined into one adder, and multiplexers can be put on the inputs to that adder. Another simplification would be to always load the μPC latch from the output of the adder, which would allow the removal of the multiplexer on the input to the latch. The only operations that were sacrificed in making the simplifications involved loading the μPC stack with the output of an adder. Figure 10 shows the block diagram of our simplified controller, and Table 2 lists the operations it performs. Notice that the controller's instruction is now broken into five fields, controlling the μPC sources, the μPC stack sources, the counter operation, the condition selection, and the μPC stack operation.

Now we will develop the geometrical and topological arrangement of the controller's subsystems.

Such arrangements are often called floor plans. A translation of the preceding ideas into the starting floor plan of the controller is shown in figure 11. The plan is composed of subsystems built of horizontal bit slices which are then stacked vertically. The number of bit slices is equal to the microcode address width for the machine, which in this case is 12 bits.

The following points were considered when deciding upon the basic framework of the floor plan. First, the μPC latch is placed adjacent to the microcode memory address pins. This is done to minimize the delay when driving addresses to the memory, as this operation is in the critical timing path for the entire machine. The input of the latch comes only from the output of the adder, so the adder should logically be placed next to the μPC latch. The adder is considerably simpler than the full arithmetic logic unit used in the data path. However, it employs the same principles as the ALU: the Manchester carry chain, the insertion of double inverters every four bits to minimize the delay in the carry chain, and the logic block to implement the desired functions with the minimum delay and power. The multiplexer is placed adjacent to the left side of the adder. This multiplexer operates in the same manner as the input multiplexer to the ALU in chapter 5. The μPC stack is then placed to the left of the multiplexer.

The only problem with this arrangement of the floor plan is that the microcode bus and the data path bus must also connect to the multiplexer. A large amount of area would be wasted if these two buses connected to the multiplexer from the side. Instead, if the buses could be placed where the μPC stack is located, they could connect to the loop counter circuits directly. But then there is the problem of where to place the μPC stack. One solution is to run the buses ***through*** the μPC stack! Each cell of the stack thus has the two buses designed right in. The two buses could then run on through the loop counter stack to the loop counter decrementer and the pads.

Having placed the major blocks of the chip into the floor plan, the layout of the control circuits can be examined, and a detailed floor plan worked out. Each of the stacks require push and pop drivers, as discussed in Chapter 3. As in the chapter 3 example, one set of drivers is placed along the top, and the other set along the bottom of the stack. The control drivers for the latch, adder, multiplexer, and counter are identical to those discussed in Chapter 5. The control bits for these control drivers could all be derived directly from the outputs of the microcode memory, but this technique would result in an exceedingly wide microinstruction. By encoding the operations to be performed by the adder and its input multiplexers, the width of the microinstruction can be dramatically decreased. With proper encoding of these operations, the functional capability of the chip is not impaired, since a number of possible control signal combinations are in fact illegal and

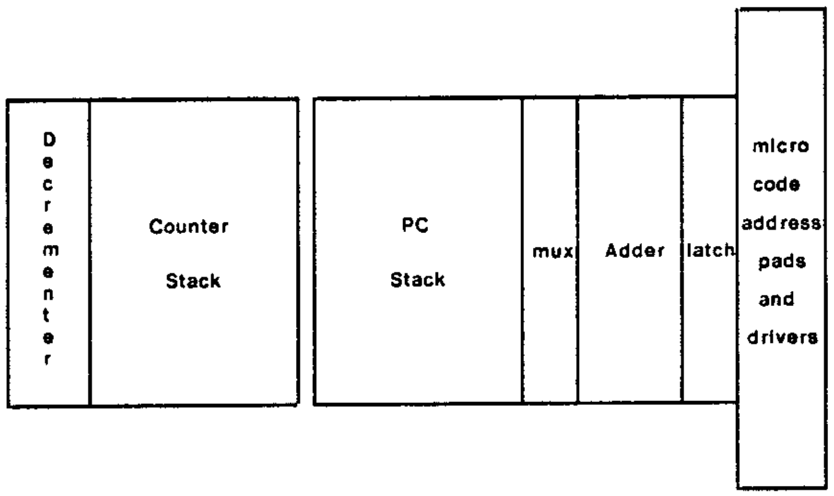


Fig.11. Starting Floor Plan for the Controller

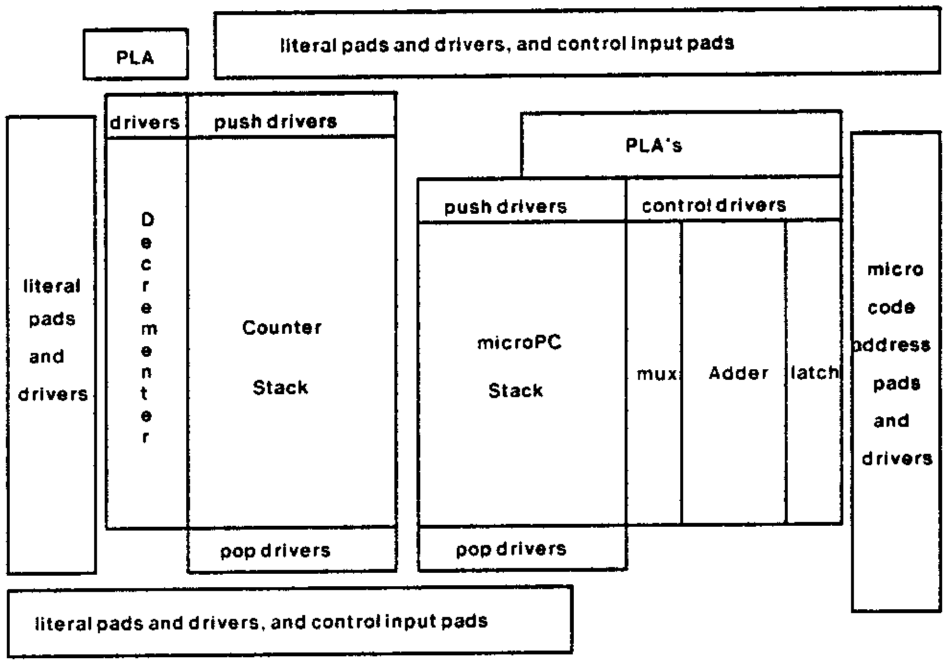


Fig.12. Final Floor Plan of the Controller Chip

thus redundant. For instance, if more than one control line for the multiplexer is enabled, the outputs of two or more sources would be shorted together, and the resulting multiplexer output would contain erroneous data. The placements of the control circuits and encoding PLAs are shown in figure 12, which also shows additional details of the final floor plan. Notice that the counter stack is *higher* than the 12-bit high μPC stack, so that it can contain entire 16 bit data path words for parameters passed to subroutines in the microcode. The stacks are aligned on their least significant bit position, and the additional length of the counter stack allowed space for the control PLAs for the adder and μPC stack.

The programmable logic arrays employed in instruction decoding do not have feedback from their outputs back into their inputs. Their only function is to serve as combinational logic for condensing the number of control wires and thus saving microcode memory bits. The *finite state machine* for the control of this path is made up of the microcode memory address feedback through the adder and stack PLAs and also the microcode literal path feedback into the input of the adder. If there were feedback terms in any of the PLAs, provisions would have to be made for access to the state of the feedback terms from off chip. Without such access, the *untestable state* information on the chip would make the testing of the completed chip next to impossible: the current operation of the chip would be a function not only of the control signals and data that we supply to the chip at a particular moment, but also of the past control signals and data. In the absence of a practical way to directly probe all the signal lines on the chip, it is imperative that all of the chip's state be accessible somehow from off chip.

One of the problems encountered in many multichip microprogrammed machines is that a great deal of interchip communication is required in their operation. Although the bandwidth of the machine can be made large by *pipelining* the operations, any operation which requires the full circle through the feedback loop of the state machine will require a great deal of time for its execution. In the OM2 system, hardware features have been included in both the data path and the controller to reduce the chip-to-chip communication as much as possible. As already mentioned, the loop counter circuitry was included on the controller chip, which reduced the loop operation time from 3 cycles to 1 cycle. Chapter 5 mentions the *conditional ALU operations* in the data path which can modify the actual ALU operation as a function of the *flag bit*. An example of the utility of this capability is provided by the multiply operation. When performing a multiply, the ALU should either add two numbers or just pass one of the numbers straight through, depending upon the state of a flag. One way to do this operation would be to send the flag to the controller chip, and execute a conditional branch to one of two locations. One of the two appropriate ALU operations would be at each of the two microcode locations. However, it would take several cycles to perform each step of a multiply were this method used. Since in OM the ALU on the data chip has the capability of modifying its instruction as a function of the flag, it actually takes only one cycle to perform this part of the multiply step.

There are times when it would be convenient to communicate many bits between the controller and the data path in one cycle. For instance, when emulating the instruction set of a higher level machine, the data path can examine various fields in the instruction currently being emulated and calculate microcode branch locations. It is then necessary to load the μPC latch with the calculated branch location. To facilitate this loading, a 16-bit bus connects the two chips, and is referred to as the "literal bus". To economize on the data path's pin count, when this bus is not transferring literal data between the two chips it is used to load microcode into the data path chip. A large number of pads are required for the microcode and data path literal interconnections. There was insufficient space along the left edge of the chip for all of the pairs of pads required for this communication. Hence, some pads were placed along the top of the chip and others along the bottom, and connections between these pads and the buses were made by running vertical wires to the appropriate bus lines where they run between the two stacks.

The layout of the completed controller chip is shown in figure 13. A floor plan of the controller is given in figure 14, for use as an aid in studying the layout figure. Examples of the use of some of the controller's operations are given in the following section.

Examples of Controller Operation

This section will illustrate the operation and programming of the controller presented in the last section through the use of four programming examples: subroutine linkage, For-loops, Do-loops, and field dispatches. Refer to Table 2 for a tabulation of the controller's opcodes. It should be noted that the μPC operations are pipelined by one cycle so that if one particular microinstruction contains a controller jump opcode, the following microinstruction will also be executed before the jump actually occurs.

To call a subroutine, we would like to save the current value of the μPC on the μPC stack and load the μPC latch with the microcode address of the subroutine. When we have finished executing the subroutine code and wish to return, we just pop the return address off of the μPC

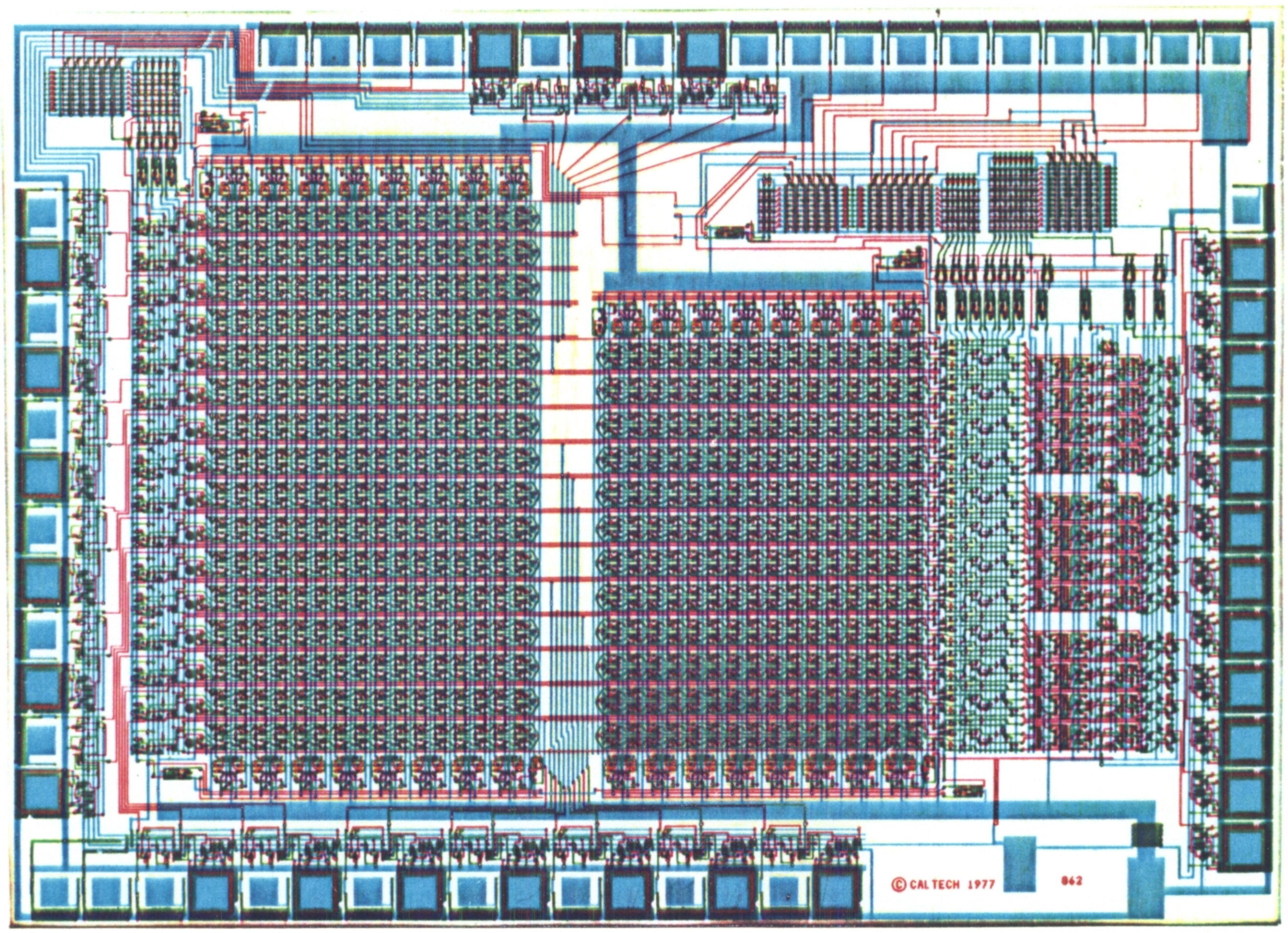


Fig.13. The Layout of the Controller Chip

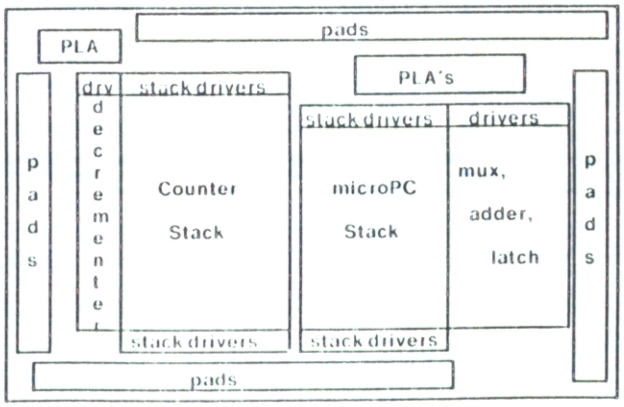


Fig.14. Map of the Controller Chip

stack and load it into the μPC latch. To save the μPC value on the stack, "μPC" should be selected as the stack source and "PUSH" should be selected as the stack operation. As Table 2 shows, the *condition* must be true in order for the stack to push a value. Therefore, the condition selection should be "TRUE" to guarantee that the stack will save the return address. While we are saving the current μPC value on the stack, we must also load the μPC latch with the subroutine address. To do this, we select "MICROCODE" as the μPC source and put the subroutine address in the literal field of the microcode. Since we are not using the counter, the counter operation should be "NOP". For the return, we load the μPC latch with the return address by selecting "STACKTOP+1" as the μPC source and pop the stack by selecting "POP" as the μPC stack operation. In order to guarantee that the stack pops the old value off the stack, we must make sure the condition is true by selecting "TRUE" as the condition selection.

Figure 15 illustrates the execution of subroutine linkages. Four "snapshots" of the microcode and μPC circuits are shown at the various steps as the execution proceeds. Snapshot (a) gives us a background for what is happening: The μPC is stepping through a segment of microcode, and is about to execute a CALL operation. The CALL operation contains a pointer to a sub-program located somewhere in the microcode memory. Snapshot (b) shows the state of the machine just after the CALL operation is executed. The μPC now points to microcode addresses inside the subroutine, while the return address to the main "program" is saved on the stack. Snapshot (c) shows that the μPC has advanced to the end of the subroutine, and the RETURN operation is about to be executed. The return address is popped off of the stack and loaded into the μPC latch, and program execution resumes where it left off in the main program, as shown in the last snapshot.

A For-loop should execute the same section of code many times. We can use the loop counter to store the number of times we have executed the code so that we know when we have finished the specified number of executions. Thus, when starting a For-loop, we should push the repetition number onto the loop counter stack. At the end of the loop we decrement the count, and if the result is not zero we should jump back to the start of the loop. If the decremented result is zero, we have finished execution of the For-loop, and we should pop the count off of the loop counter stack. Executing the For-loop in this manner requires that the end-of-loop command contain the address of the start of the loop. How then can we construct relocatable code containing For-loops? We can eliminate the need for the end-of-loop command to contain the loop's start address, by saving the start address on the μPC stack. The μPC latch would just have to be loaded with the value contained at the top of the stack. Using this method of saving the loop address, the start-of-loop command becomes:

```
μPC Source ← μPC + 1
μPC Stack Source ← μPC
μPC Stack Operation ← Push
Condition ← True
Counter Operation ← either Push Microcode or Push Literal
```

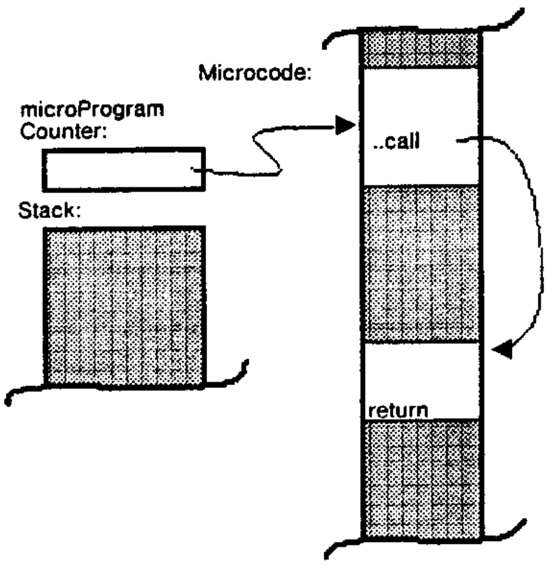
and the end-of-loop command becomes:

```
μPC Source ← Stacktop + 1
μPC Stack Operation ← Pop
Condition ← Count NOT EQ 0
Counter Operation ← True: decrement; False: Pop
```

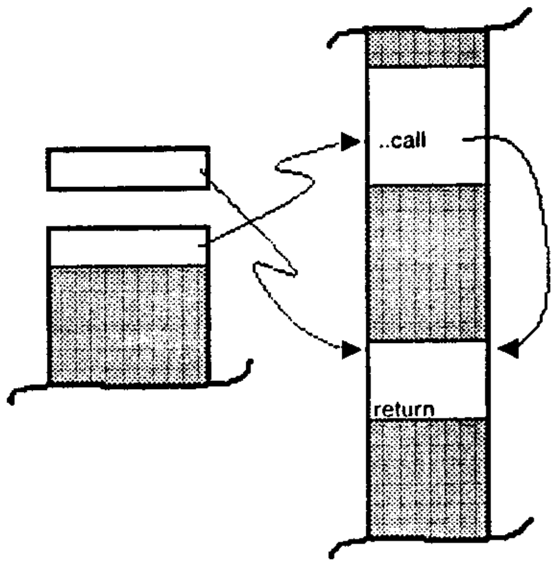
The operation of For-loops is illustrated in figure 16. Again, four snapshots are shown which represent the state of the controller and microcode at various points in the execution of the loop. Snapshot (a) shows the state of the machine just prior to the execution of the FOR operation. When the FOR operation is executed, the value in the μPC latch is pushed onto the μPC stack, and the number of iterations specified by the FOR command is pushed onto the counter stack. The μPC continues advancing through the microcode. Snapshot (b) shows the state of the controller and microcode at some point in the middle of the FOR loop execution. When the end of the loop is reached, the value on the top of the counter stack is decremented. If the result is not zero, the new value is pushed onto the stack and the μPC latch is loaded with the value on the top of the μPC stack, as shown in snapshot (c). Notice that the value is not popped of the top of the μPC stack, because we will need the loop address again if the loop is not completed after executing one more time. When the result is zero, data is popped off the top of both stacks (to remove the loop address and the old count, which is now =0) while the μPC value is just incremented, causing the controller to exit from the FOR loop, as shown in the last snapshot.

The Do-loop is similar to the For-loop, except that the code is repeatedly executed until a condition becomes true. That condition may be, for instance, when the data path flag becomes true. In this case, the condition selection in the end-of-loop command becomes "DATA PATH FLAG" instead of "COUNT NOT EQ 0". Also, since the counter is not being used, the counter operation in both the start-of-loop and the end-of-loop commands becomes "NOP".

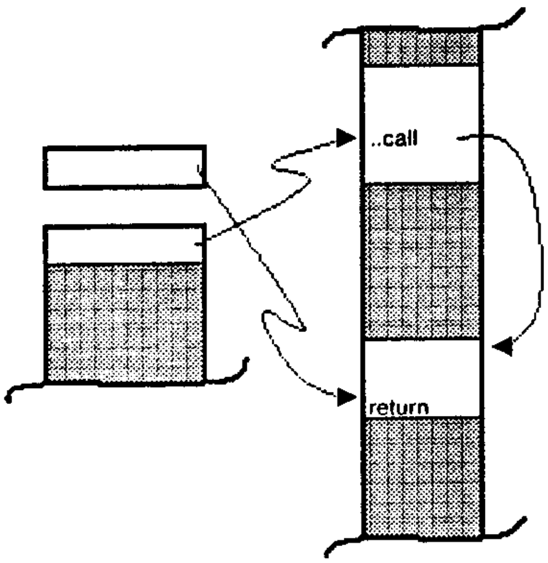
Figure 17 shows some snapshots associated with the execution of a DO loop. By comparing figures 16 and 17, the similarities between FOR loops and DO loops can be observed. Basically,



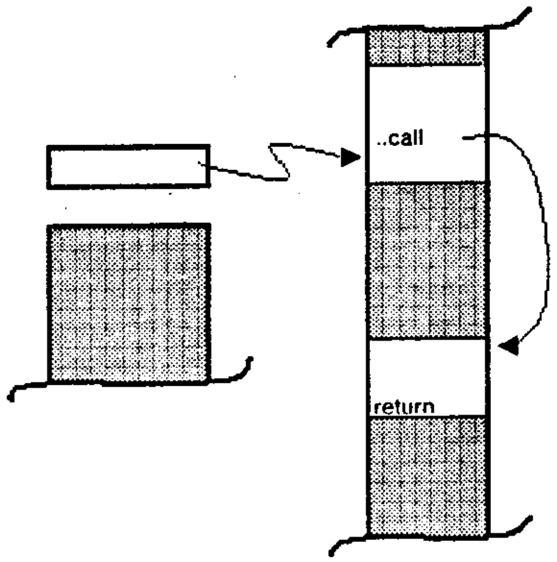
(a) Before Execution of the "Call" Instruction.



(b) Just After Execution of the "Call" Instruction.

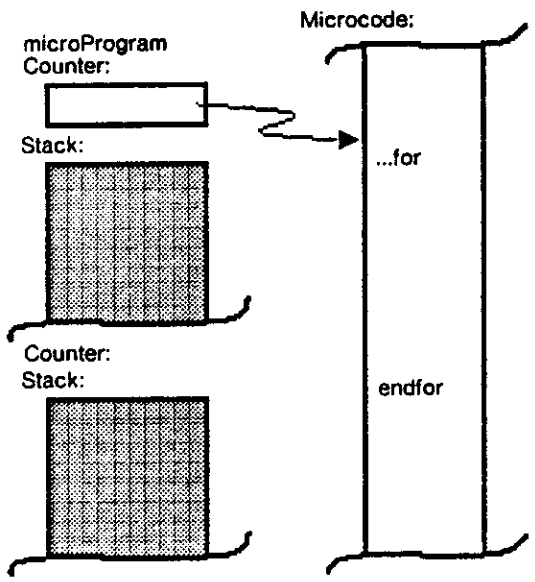


(c) Just Before Execution of the "Return" Instruction.

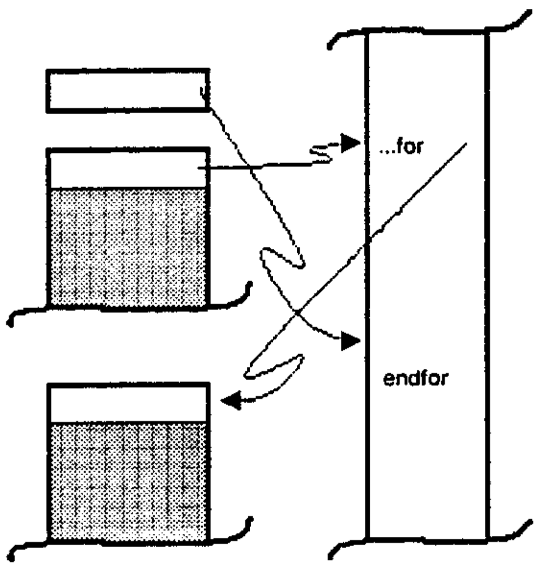


(d) After Execution of the "Return" Instruction.

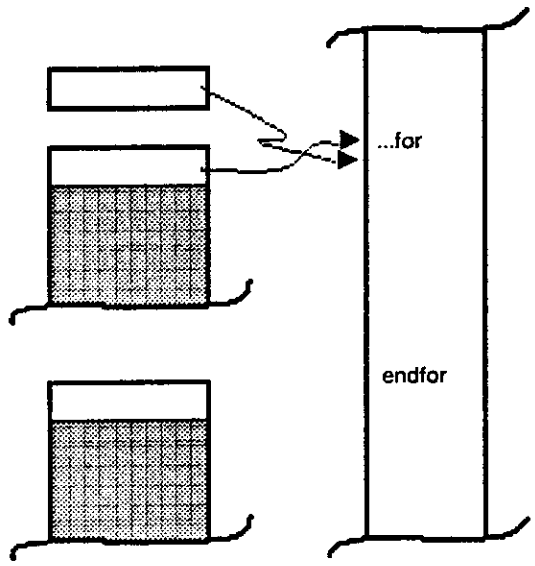
Fig.15. Illustration of Subroutine Linkage.



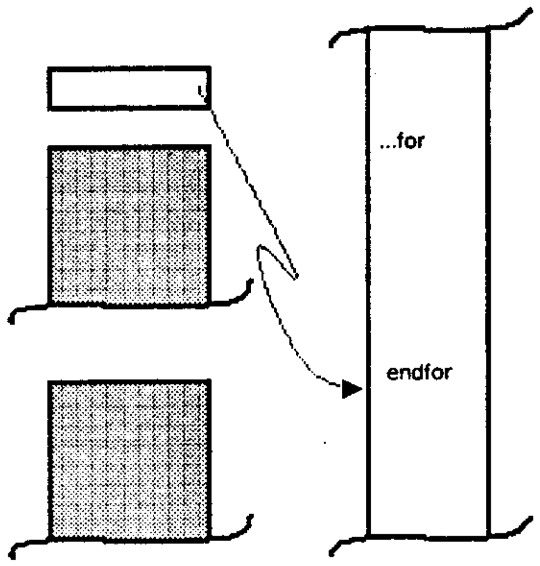
(a) Before Execution of the "For"



(b) Execution of the "For" loop.

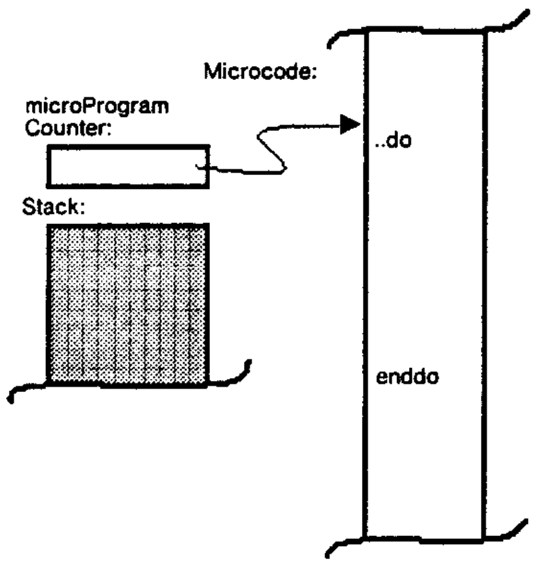


(c) Repeat Execution of the "For" loop when count is not zero.

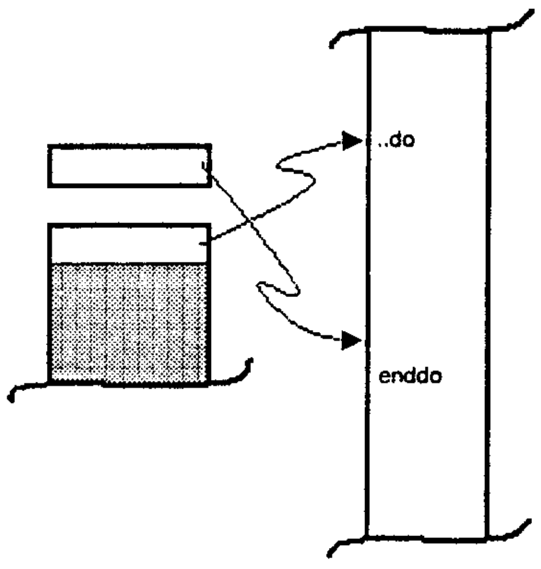


(d) Exit Loop when Count is Zero.

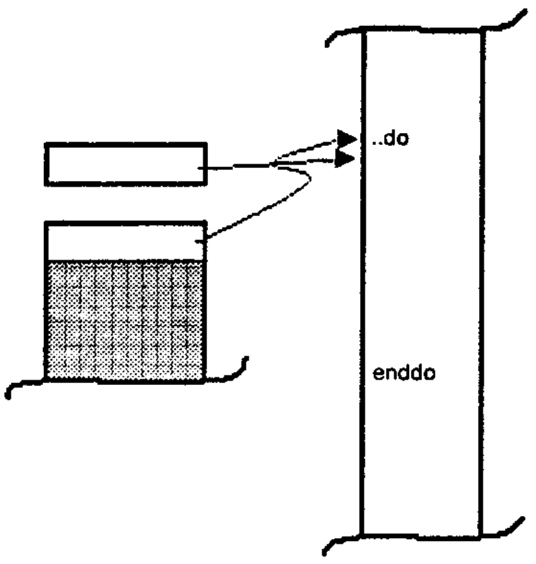
Fig.16. Illustration of the Operation of "For" Loops.



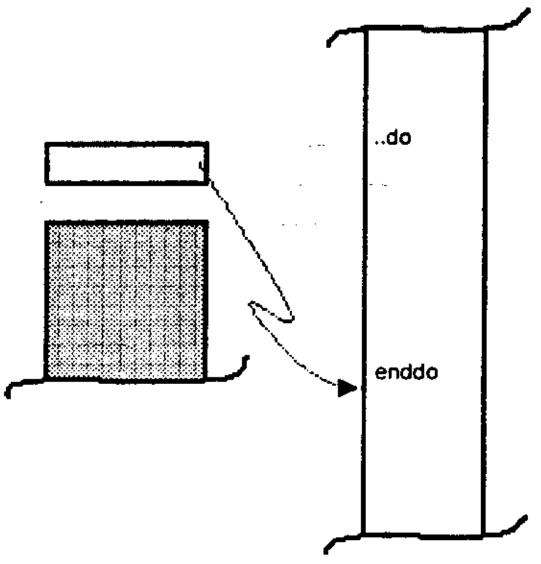
(a) Before Execution of the "Do" Instruction.



(b) Execution of the "Do" Loop.



(c) Repeat Execution of the "Do" Loop While Condition is False.



(d) Exit "Do" Loop When Condition is True.

Fig.17. Illustration of the Operation of the "Do" Loop.

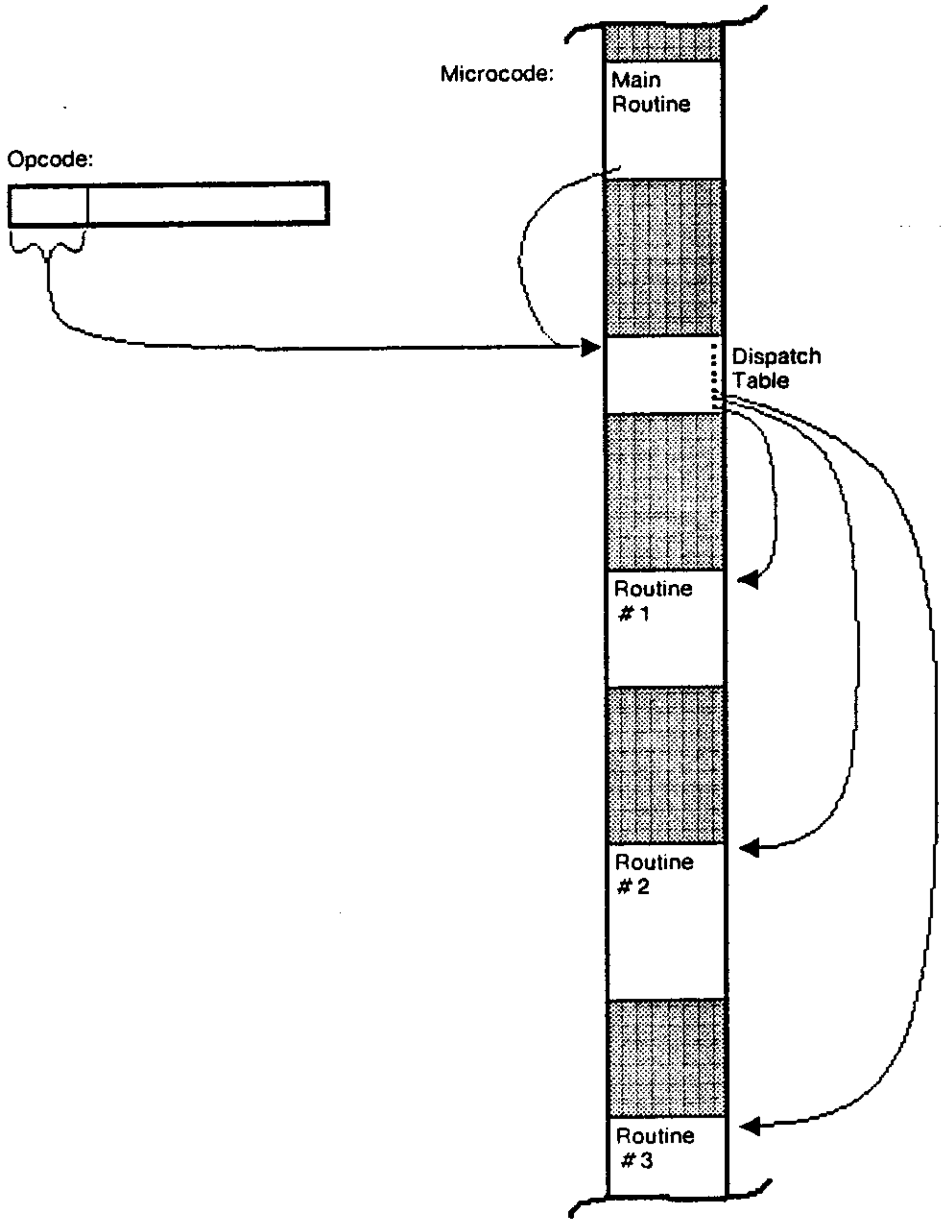


Fig.18. Illustration of the Operation of the "Dispatch" Instruction.

the only difference between these two types of loops is the decision of when to exit the loop. In a FOR loop a counter decides when the loop should be exited, while in the DO loop a flag, such as the flag from the Data Path, decides when the loop should be exited. Since the DO loop does not use the counter, the counter is not shown in the snapshots of figure 17.

When emulating the instruction set of a higher level machine, it is often convenient to do a multi-way branch. Suppose, for example, that the machine we are emulating has a 16-bit instruction word that contains a 4-bit opcode field and a 12-bit address field. In this case, we would have 16 code segments in the microcode, one for emulating each of the 16 possible opcodes of the higher level machine. We would like to be able to perform a 16-way branch, depending on the contents of the 4-bit opcode field, that would take us directly to the correct microcode segment, thus implementing the *decode stage* of instruction interpretation. We could use the ALU in the data path for calculating the microcode address for the proper segment, and load the μPC latch with the result of this calculation. This works especially nicely if the starting addresses of the segments are evenly spaced, because to calculate the branch address we merely multiply the 4-bit opcode by the segment length and add the displacement of the first segment. The multiplication is particularly easy to perform if the segment length is a power of 2, because then we just have to shift the 4-bit opcode value the appropriate number of places to the left.

A problem with the above method of field dispatching is that the microcode segments have to be evenly spaced in the microcode, preferably by a power of 2. In practice, segments are seldom of the same length. Even if they were of the same length, if one of the segments had to be modified, extensive corrections might have to be made all through the microcode. As an alternative, a *dispatch table* can be inserted into the microcode, which just contains a series of jump instructions to the appropriate microcode segments. If this is done, the 4-bit opcode value need only be shifted left once (because jump instructions are two microcode words long due to pipelining), added to the dispatch table displacement, and loaded into the μPC latch. To load the value into the μPC latch, the data path sends the result of the above calculation across the literal bus to the controller, and the controller selects a μPC source of "LITERAL".

Figure 18 illustrates the operation of the dispatch instruction. The controller jumps to a location in the dispatch table that is a function of one of the fields in the opcode. The dispatch table contains JUMP instructions to the various routines that perform the micro-instructions necessary to emulate each of the possible opcodes. The selection of the proper field in the opcode and the calculation of the dispatch table address are performed in the Data Path prior to the dispatch.

Some Reflections on the Classical Stored Program Machine

In the future, very large quantities of computing machinery may be placed on a single chip. Such chips will be easily and quickly designed, and rapidly implemented. This capability will present both a great opportunity, and a great challenge. How are we to organize and program such a wealth of hardware? Certainly not the way we do now.

Scaling of the technology to higher densities is producing effects which may be clarified by analogy with events in civil architecture. Decades ago, standard bricks, "two-by-fours", and standard plumbing were used as common basic building blocks. Nevertheless, architects and builders still explored a great range of architectural variation at the top level of the time: the building of an individual home. Today, due to the enormous complexities of large cities, many architects and planners have moved on to tackle the larger issues of city and regional planning. The basic building blocks have become the housing tract, the business district, and the freeway network. While we may regret the passing of an older style and its traditions, there is no turning back of the forces of change.

In present LSI, where we can put many circuits on a chip, we are like the earlier builder. While we no longer tend to explore and locally optimize at the circuit level (the level of bricks and two-by-fours), we still explore a great range of variation at the level of the individual computer system. In future VLSI, where we may put many processors on a chip, architects will, like the city planner, be more interested in how to interconnect, program, and control the flow of information among the components of the overall system. They will move on to explore a wider range of issues and alternatives at that level, rather than occuping themselves with the detailed internal structure, design, and coding of each individual stored program machine within a system. If systems are to work at all, they must at the least be understood at their highest level. These are some of the issues explored in chapter 8.

References

1. A. W. Burks, H. H. Goldstine, J. von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument", Institute for Advanced Study report to the Army Ordnance Dept., 1946; contains the original description of a stored program electronic digital computer. Reprinted as *Chapter 4* in Bell and Newell[6].

2. A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem", Proc. of the London Mathematical Society, Series 2, Vol.42 (1936-37), pp.230-265.

3. M. V. Wilkes, "The Best Way to Design an Automatic Calculating Machine", address to the Manchester University Computer Inaugural Conference, July 1951. The basic principles of microprogramming were first stated by Wilkes in this address.

4. M. V. Wilkes, J. B. Stringer, "Microprogramming and the Design of the Control Circuits in an Electronic Digital Computer", Proc. Cambridge Phil. Soc., pt. 2, vol.49, pp. 230-238, April, 1953. An extension of the earlier work by Wilkes, this paper includes a worked out example. Reprinted as *Chapter 28* in Bell and Newell[6].

5. P. M. Davies, "Readings in Microprogramming", IBM Systems Journal, No.1, pp. 16-40, 1972., Provides a good primer and guide to the literature on microprogramming.

6. C. G. Bell, A. Newell, "Computer Structures: Readings and Examples", McGraw-Hill, 1971, presents some of the history of computer architecture and design, and contains many interesting examples of computer structures.

7. D. L. Dietmeyer, "Logic Design of Digital Systems", chapter 6, Allyn and Bacon, 1971, contains an example of the design of an elementary digital computer, starting with machine instruction set and controller state diagram.

8. C. R. Clare, "Designing Logic Systems Using State Machines", McGraw-Hill, 1973.

9. D. L. Johannsen, "Our Machine: A Microcoded LSI Processor", Display File #1826, July 1978, Dept. of Computer Science, California Institute of Technology.