## 5.3 SYSTEM OVERVIEW

An informal block diagram of one OM system is shown in Fig. 5.1. Such a system is a complete stored-program, general-purpose computer. Input/output devices are usually interfaced via the external data bus and control lines, located to the left in Fig. 5.1. Several such systems may be interconnected via the system bus to augment one user's overall system. Tasks may then be distributed among the OM systems, for example, using different ones to independently control different input/output devices, thereby improving overall system performance. Groups of different user systems may also share the system bus.

Each OM system is composed of five LSI chips, along with some standard memory chips and a few MSI chips. A brief description follows of the five LSI chips being designed as part of this project.

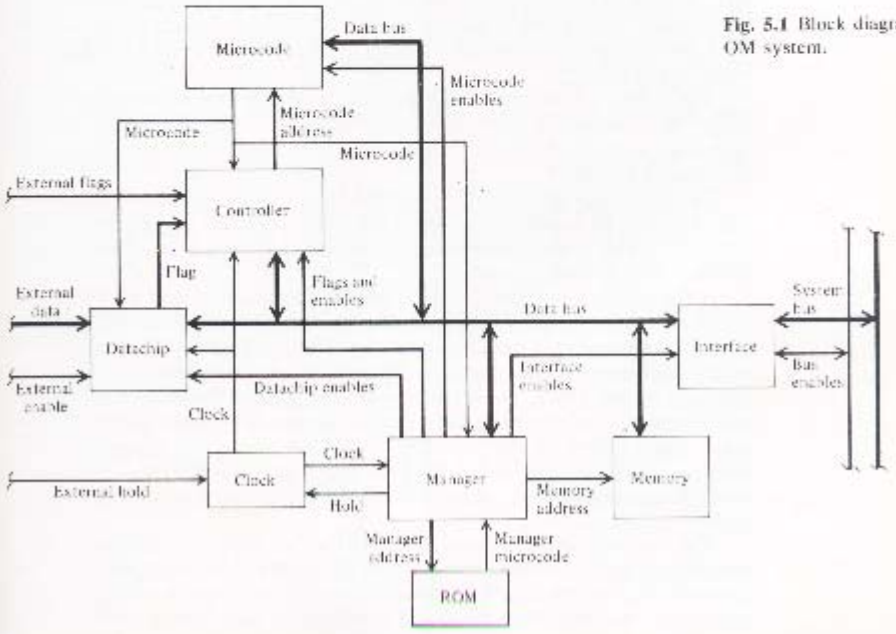



Fig. 5.1 Block diagram of an OM system.

The *data path chip* performs most of the data manipulation functions for the system. The operations are performed as directed by sequences of control microinstructions, which are fetched from a microcode memory using addresses generated by the controller chip. The main subsystems of the data path chip are a register array, a shifter, and an arithmetic logic unit (ALU). Two buses connect these subsystems. This chip's internal structure is described in detail later in this chapter.

The *controller chip* contains the microprogram counter ($\mu$PC) that stores the microcode memory address, and a counter for the control of microprogram loops. The chip also contains stacks for both the microprogram counter and loop control counter values. The concepts of controller structure and function are fundamental in computer architecture. (Chapter 6 provides an introduction to these ideas and then describes the organization and layout of the controller chip.)

The *memory manager chip* provides addresses for the data memory and directs the communication between chips on the data bus. It also implements some simple data structures in the data memory. The manager can divide the memory into separate partitions and implement a different data structure in each partition. Four basic data structures are implemented: stacks, queues, linked lists, and arrays. When accessing a stack partition, for example, the microcode need only ask the manager to push or pop data on or off the stack. The manager maintains the stack pointers, performs bounds checking to see if the stack is full or empty, and transfers the data.

The *system bus interface chip* provides asynchronous communication with other OM systems via the system bus. There are a whole host of subtleties associated with interfacing asynchronous buses. These issues are among those discussed in Chapter 7.

The *clock chip* generates the two-phase clock signals needed by the system. The clock can be stopped to allow for the synchronization of asynchronous signals. Some chips in the system have only a single clock input; those chips generate the two clock phases on-chip.

A few words about timing may be helpful: In general, during $\varphi_1$ data bits are transferred from one subsystem to another on the same chip, while during $\varphi_2$ data bits are transferred from one chip to another. The data chip's ALU, and other data modification units, operate during $\varphi_2$. Microcode is available on both phases and is pipelined by one phase. Thus, the opcodes that control the ALU enter the data chip during $\varphi_1$. The microprogram address is generated by the controller chip during $\varphi_2$, gets driven off-chip into the data chip's microcode memory's latches during $\varphi_1$, and is used to look up the next opcode on the following $\varphi_2$. Because of these timing requirements, all jumps in the microcode are pipelined by one clock cycle.

The remainder of this chapter describes the data path chip and is presented in two distinct parts. The first part outlines the architectural requirements for the data path chip and then illustrates, via the detailed design and layout of the chip's subsystems and cells, how the design methodology was applied to satisfy these

requirements. The second part is an external functional description of the data path chip, intended as a user manual for those who microprogram the computer system, and for reference while studying the OM2 controller chip in Chapter 6.

## 5.4 THE OVERALL STRUCTURE OF THE DATA PATH

The basic requirements initially established for the data path chip were (1) that it be gracefully interconnectable into multiprocessor configurations, (2) that it effectively support a microprogrammed control structure, thus enabling machine instruction sets to be configured to the application at hand, (3) that it be able to do variable field operations for emulation instruction-decoding, assembly of bit-maps for graphics, etc., and (4) that its performance be as fast as possible.

In order to satisfy the first requirement, the data path chip was designed with two ports: one port to be used for a system interconnection, and the other for connection to local memory, input/output devices, etc. The requirement for gracefully handling variable-length words required a shifter at least sixteen bits long. The performance requirement dictated an arithmetic logic unit having considerable flexibility without sacrificing speed. In many systems time is lost in assembling the two operands required for most operations. Therefore, the data path has two internal buses, and all registers on the chip are two-port registers. In order to avoid extensive random wiring for connecting the major subsystems on the chip, the following strategy was adopted at the outset: two internal buses would run through the entire processing array, from one end of the chip to the other. One port was to be located at the left end of the chip, and the other port at the right end.

The three main functional blocks on the chip are the arithmetic logic unit, the shifter, and the register array. These blocks are placed next to each other in the center of the chip, between the two ports. The arrangement of the major subsystems is shown in Fig. 5.2. The system buses run horizontally, on the polysilicon level, through these functional blocks. The major control lines run vertically across these blocks, on the metal level. The power, ground, and clock lines are run parallel to the control signal lines. The details of these functional blocks will be

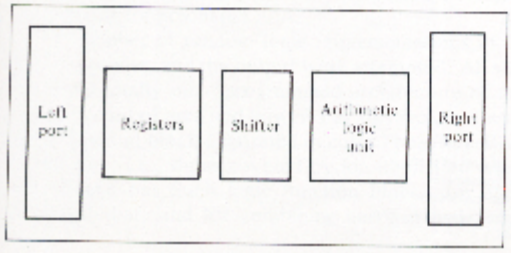



Fig. 5.2 General floor plan of the data path chip.

described in subsequent sections of this chapter. Included are descriptions of peripheral circuits needed to interface subsystems with each other and to the outside world. Detailed layouts of certain cells in the system are also included. The overall layout of the data chip is shown in the frontispiece.

### 5.5 THE ARITHMETIC LOGIC UNIT

The carry chain of the ALU, and its associated logic, was the first functional block to be designed in detail, since it was believed that the carry chain would limit the performance of the system. Simulations of several look-ahead carry circuits indicated that they would add a great deal of complexity to the system without much gain in performance. For this reason a decision was made early in the project to implement the fastest possible Manchester-type carry chain, [Chap. 1, 5.8] having a carry propagation circuit similar to that shown in Fig. 1.28. The carry chain and its associated logic were allowed to dictate the repeat distance of the cells in the vertical direction. In *n*MOS technology, a Manchester carry chain is particularly limited in its ability to propagate a *high* carry signal. However, it can quite rapidly propagate a *low* carry signal.

In the arithmetic logic unit there will be a null period when the OP code for the next operation is being brought in. Advantage can be taken of this null period to precharge the carry chain and other sections of the data path where timing is particularly crucial. In this way, it is not necessary to propagate high signals through pass transistors where the rise transient would be particularly slow. This strategy was applied in OM's ALU; the resulting carry chain is shown in Fig. 5.3.

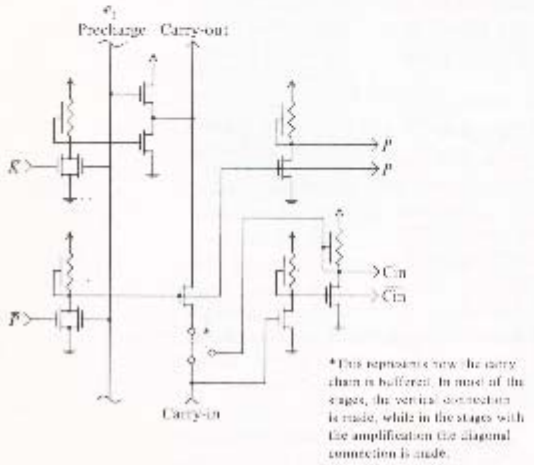



**Fig. 5.3** Carry-chain circuit for the arithmetic logic unit.

The main carry chain runs through a pass transistor from carry-in to carry-out. The carry-in signal is detected by the gate of an inverter that feeds the signal into the subsequent logic of the ALU. Three transistors are used to control the state of the carry-out of each stage. The first one merely precharges the node associated with carry-out during the null period of the ALU. The second is the carry-kill signal that is derived from the inputs to the ALU, and it simply grounds the carry-out through a single transistor. The third is the pass transistor that causes carry-out to be equal to carry-in. These last two signals associated with the carry chain in each stage, carry-kill and carry-propagate, are generated by two NOR gates that have kill-bar and propagate-bar as one input and precharge as the second input. Hence, it is assured that the kill signal and propagate signal are disabled during the null period when the precharging takes place.

After some analysis, we found that nearly all interesting combinations of carry-in and the input signals could be generated using propagate and carry-in from each stage. Thus, as in Fig. 5.4, the carry chain may be seen as a logic block with two inputs, carry-kill and carry-propagate; the outputs, propagate and carry-in; vertical signals, carry-in and carry-out; and one control wire, precharge.

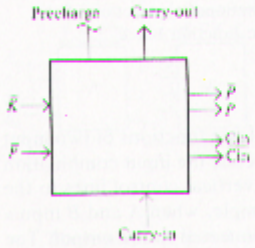



**Fig. 5.4** Abstraction of the carry-chain circuit.

The task of designing the balance of the ALU is now reduced to that of designing functional blocks to (a) combine the two input variables to form a propagate-bar and kill-bar, and (b) combine carry-in and propagate to form the output signal, and then designing drivers for controlling the logic function blocks and deriving a timing for precharge.

A number of random logic implementations of function blocks for deriving kill, propagate, and the output were attempted. All seemed to be at variance with the horizontally microprogrammed architecture of the data path and required a large amount of area and power. For this reason it was decided to use the general logic function block illustrated in color Plate 7(a). Recall that the depletion mode transistors, i.e., those covered by ion implanted regions represented by yellow, are always on. Such logic function blocks are used to generate kill-bar and propagate-bar, and for combining carry-in and propagate to form the output.

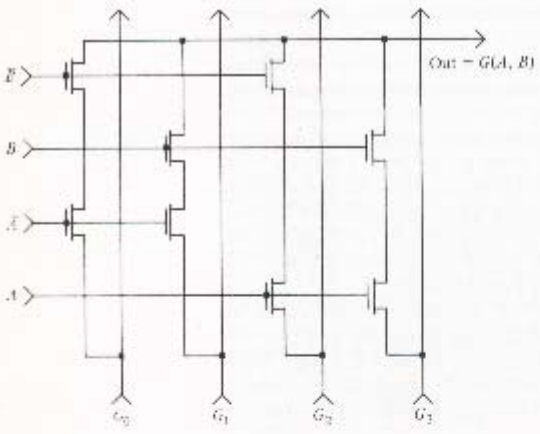



Fig. 5.5 General logic function block diagram for a transistor.

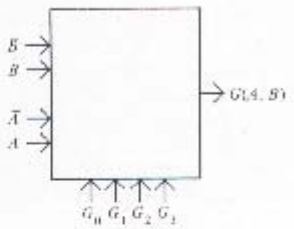



Fig. 5.6 Functional abstraction of the general logic function block.

The circuit, shown in Fig. 5.5, implements the sixteen logic functions of two input variables. It consists of a set of transistors that fully decode the input combination of $A$ and $B$. The set connects one and only one of the vertical control lines to the output, depending on this input combination. For example, when $A$ and $B$ inputs are both low, the vertical control wire labeled $G_0$ is connected to the output. The truth table entries for the desired logic function are placed on the $G$ vertical control wires, and the output is then the desired logic function of the two input variables. For example, if the Exclusive-OR of $A$ and $B$ is desired, a logic-0 will be applied to the control wires $G_0$ and $G_3$, and logic-1 will be applied to control wires $G_1$ and $G_2$. Since it is desired to implement the same logic function on all bits of the word, the control variables $G_0$ through $G_3$ need not be generated in every bit slice, but may be generated once at either the top or bottom of the array. The functional abstraction of the circuit of Fig. 5.5 is shown in Fig. 5.6. For a color-coded stick diagram of the function block, see Plate 10(a). Plate 10(b) shows a color-coded actual layout of the function block.

The block diagram for a complete arithmetic logic unit is shown in Fig. 5.7. The functional dependence of the output on the two inputs and the state of the carry is determined by a 12-bit number: $P_0$ through $P_3$, $K_0$ through $K_3$, and $R_0$ through $R_3$, together with the carry-in to the least significant bit of the ALU. The ALU is quite general, and its detailed operation set may be left unbound until the control structure of the computer system is designed.

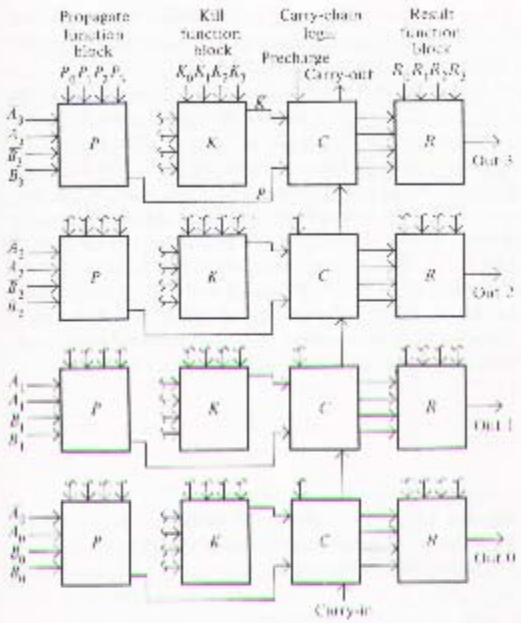



**Fig. 5.7** Block diagram of a 4-bit ALU.

There are two general principles illustrated by this design. (1) It is often less expensive in area, time, and power to implement a general function than to implement a specific one. (2) If a general function can be implemented, the details of its operation can be left unbound until later, and hence provide a much cleaner interface to the next level of design. The detailed selection of which functional entities to leave unbound and which to bind early requires a considerable amount of judgment.

Two details must be dealt with before the arithmetic logic unit subsystem is complete. Drivers are needed for the $P_0 \ldots P_3$, $K_0 \ldots K_3$, and $R_0 \ldots R_3$ control lines that will generate signals with the appropriate timing. In addition, inverters must be interposed in the carry chain occasionally to minimize the propagation delay through the entire carry chain. The way we have chosen to implement the interposition of inverters is to recognize that each carry chain function block contains two inverters that produce at their output the carry-in, having been twice inverted from the actual carry-in signal. If we merely substitute this signal for the carry-in signal to the pass transistor, we have doubly inverted our carry-in and buffered it to minimize the propagation delay. This approach avoids putting spaces

for inverters between the carry function blocks. It is illustrated by the dotted connection lines in Fig. 5.3. In the actual implementation, the connection through the inverters was made in every fourth stage (see Section 1.11).

Drivers for the $P$, $K$, and $R$ control lines have the following function: At some time during the null period of the ALU (which occurs during $\varphi_1$), an OP code specifying the state of each control line arrives at the drivers. It must be latched while the ALU itself is being precharged, and then it must be applied to the $P$, $K$, and $R$ control lines as soon as the ALU is activated. The $P$, $K$, and $R$ function blocks are themselves composed of pass transistors, and their outputs are more effectively driven low than high. For this reason, we will precharge the outputs of the $P$, $K$, and $R$ function blocks as well as the carry chain itself. This is most conveniently done by requiring that all of the $P$, $K$, and $R$ control signals be high during the null period of the ALU. Then, independent of the states of $A$ and $B$ inputs, the outputs will be charged high by the time the ALU active period commences. The control driver implementing this function is shown in Fig. 5.8.

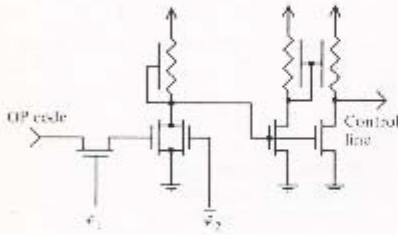



Fig. 5.8 ALU control driver. All outputs high during $\bar{\varphi}_2$; selected terms low during $\varphi_2$; OP code valid during $\varphi_1$.

The OP code is latched through a pass transistor whose gate is connected to $\varphi_1$, and the OP code runs into a NOR gate, the other input of which is also $\bar{\varphi}_2$. Thus, the output of the NOR gate is guaranteed to be low during the $\varphi_2$ period. The NOR gate output is then run through an inverting super buffer, so that during $\varphi_1$ the output is guaranteed to be high. At the end of $\varphi_1$, the OP code is driven onto the $P$, $K$, and $R$ control lines. The only interface specification for the ALU that must be passed to the next level of system design is that the $P$, $K$, and $R$ control signals be valid before the end of $\varphi_1$, and that the $A$ and $B$ inputs likewise be valid by the end of $\varphi_1$ and be stable throughout $\varphi_2$, the active period of the ALU. We are then guaranteed that after enough time has passed to allow the carry to propagate, the output of the $R$ function block will accurately reflect the specified function of the ALU and may be latched at the end of $\varphi_2$.

A color-coded plot of the layout of a 1-bit slice through this ALU is shown in color Plate 11. This ALU went through another design iteration before inclusion in the OM2. A plot of the ALU control driver is shown in color Plate 12.

## 5.6 ALU REGISTERS

In order for the arithmetic logic unit described in the last section to be useful, it must be equipped with a set of registers both for its input variables and for its output. Let us consider the input registers first. Inputs to the ALU may be derived from either the shifter, the buses, or other sources. They may be latched and left unchanged during any $\varphi_1$-$\varphi_2$ machine cycle or set of machine cycles. This is one of the situations in which combining the multiplexing function with the latching function simplifies the design and achieves better performance. A register operating in this manner is shown in Fig. 5.9.

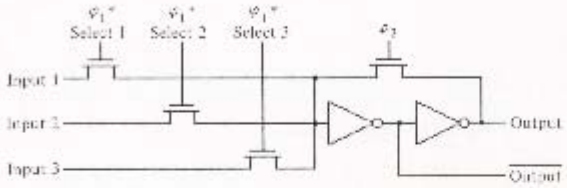



**Fig. 5.9** ALU input register and multiplexer.

The input to the first inverter can be derived from four sources: three external sources such as shifter output, bus, etc., and a fourth, the output of the second inverter. When it is desired to latch a new signal into the register, one of the source pass transistors is driven on during $\varphi_1$. The feedback transistor around the two inverters is always activated during $\varphi_2$. Thus, with three vertical control wires plus the $\varphi_2$ timing signal, it is possible to select one of three sources into the register, or none of the three sources, thereby leaving the previous value of the register stored on the gate of the first inverter during the $\varphi_1$ period. Since it is necessary to have two inverters to form the stable pair when the feedback transistor is on, both the input and its complement are available as required by the $P$ and $K$ function blocks of the arithmetic logic unit. The OP code signal that selects the source that will be applied to the ALU input register during $\varphi_1$ must enter the chip during the previous $\varphi_2$. Each of the select signals must be low during $\varphi_2$, and at most one of them may come high during the following $\varphi_1$. A driver appropriate for these control signals is shown in Fig. 5.10, with the corresponding layout shown in color Plate 12. The control OP code is latched during $\varphi_2$, during which time the NOR gate shown disables the output driver. Since the output driver in this case is noninverting, the output select line is held low during all of $\varphi_2$. At the end of $\varphi_2$, the OP code signal is latched and at the beginning of $\varphi_1$ the particular select line to be enabled that cycle is allowed to go high.

Note that this timing allows two incoming OP code bits per external wire per machine cycle. In particular, if it were desirable to share a microcode bit between the ALU function and the ALU selector inputs, this could be done by bringing the ALU OP code in during $\varphi_1$ and the ALU input selection code in during $\varphi_2$. This technique was suggested by Ivan Sutherland.

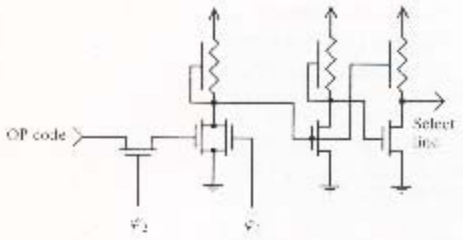



Fig. 5.10 Select control driver. All outputs low during $\varphi_1$; selected terms high during $\varphi_1$; OP code valid during $\varphi_2$.

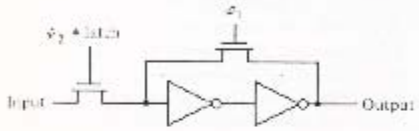



Fig. 5.11 Output register.

The ALU output register is similar to the ALU input register, except that the timing is reversed. The result of the ALU operation is available at the end of $\varphi_2$. An OP code bit will, if desired, enable the latch signal to go high during $\varphi_2$. The feedback transistor is always enabled during $\varphi_1$, and thus the latch is effectively static even though in the absence of a latching signal the data is stored dynamically on the gate of the first inverter through the $\varphi_2$ period. Once again, both the output and its complement are available if desired.

### 5.7 BUSES

An early design decision was made to have data flow through the data path chip on two buses that communicate with all of the major blocks of the system. We have already seen that the ALU performs its operation during the $\varphi_2$ period and does not have valid data to place into its output register until the end of $\varphi_2$. If data are to be transferred from the output register of the ALU to its input register, this must be done during the $\varphi_1$ period. If we adopt a standard timing scheme in which all transfers on the buses occur during $\varphi_1$, we can make use of the $\varphi_2$ period when the ALU is performing its operation to precharge the buses in the same manner that the carry chain was precharged during the $\varphi_1$ period. In this way we solve one of the knotty problems associated with a technology designed for ratio logic. If we had insisted that the tri-state drivers associated with various sources of data for a bus be able to drive up as well as down, we would have required both a sourcing and sinking transistor, together with a method for disabling both transistors. While it is perfectly possible to build such a driver (we shall undertake the exercise as part of the design of the output ports), it is a space-consuming matter to use such a driver at every point where we wish to source data onto an internal bus.

By using the bus precharge scheme, our "tri-state drivers" become simply two series transistors as shown in Fig. 5.12. Here the data from one source, for example, the ALU output register, is placed on the gate of one of the series transistors. An enable signal, which may come high during $\varphi_1$, is placed on the other series transistor. If one and only one of the enable signals is allowed to come high during any one $\varphi_1$ period, the bus can be driven from as many sources as necessary. The performance of such a bus is limited by the pull-down capability of the two series transistors. We attach such a driver to each of the output registers for the ALU.

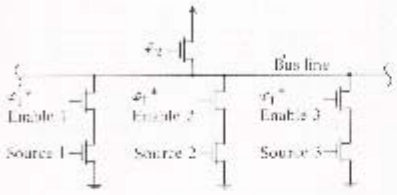



Fig. 5.12 Precharged bus circuit.

### 5.8 BARREL SHIFTER

Since shifting is basically a simple multiplexing function, one might think that a shifter could be combined with the input multiplexer to the ALU. A simple 1-bit, right-left shifter implemented in this manner is shown in Fig. 5.13. It is identical with the three-input ALU register, with the three inputs used to select among the bus, the bus shifted left by one, and the bus shifted right by one. To support the

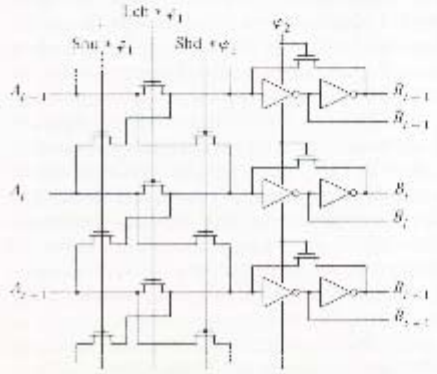


Fig. 5.13 A simple 1-bit, right-left shifter.

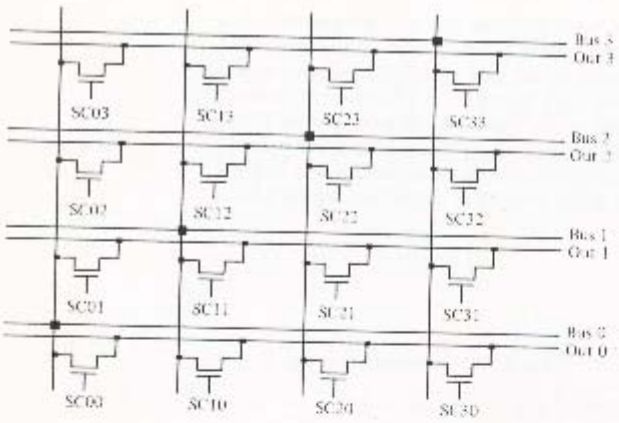



Fig. 5.14 A 4-by-4 crossbar switch.

multibit shifts necessary for field extraction and building up odd bit arrays, something more is required. One is tempted initially to build up a multibit shift out of a number of single shifts. However, for word lengths of practical interest, the $n^2$ delay problem (see Section 1.11) makes such an approach unworkable.

The basic topology of a multibit shift dictates that any bus bit be available at any output position. Therefore, data paths must run vertically at right angles to the normal bus data flow. Once this simple fact is squarely faced, a multibit shifter is seen as no more difficult than a single bit shifter. A circuit enabling any bit to be connected to any output position is shown in Fig. 5.14. It is basically a crossbar switch with individual MOS transistors acting as the crossbar points, the basic idea being that each switch $SC_{ij}$ connects bus $i$ to output $j$. In principle this structure can be set to interchange bits as well as to shift them, and it is completely general in the way in which it can scramble output bits from any input position. In order to maintain this generality, the control of the crossbar switch requires $n^2$ control bits. In some applications, the $n^2$ bits may not be excessive, but for most applications a simple shift would be adequate. The gate connections necessary to perform a simple barrel shift are shown in Fig. 5.15. The shift constant is presented on $n$ wires, one and only one of which is high during the period the shift is occurring. If the shifter's output lines are precharged in the same manner as the bus, the pass transistors forming the shift array are required to pull down the shifter's outputs only when the appropriate bus is pulled low. Thus, the delay through the entire shift network is minimized and effective use is made of the technology.

A second topological observation is that in every computing machine, it is necessary to introduce literals from the control path into the data path. However,

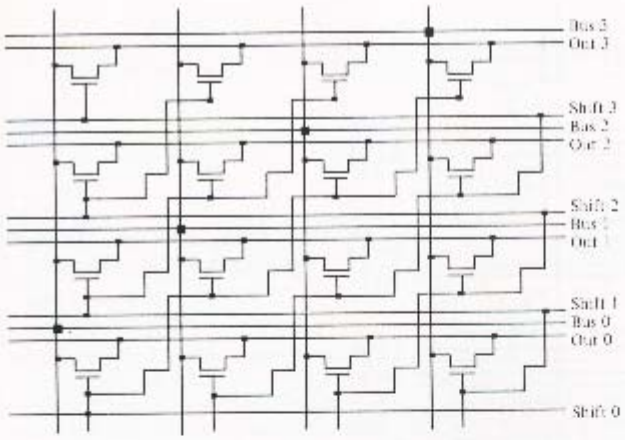



**Fig. 5.15** A 4-by-4 barrel shifter.

our data path has been designed in such a way that the data bits flow horizontally while the control bits from the program store flow vertically. In order to introduce literals, some connection between the horizontal and vertical flow must occur. It is immediately obvious in Fig. 5.15 that the bus is available running vertically through the shift array. That is then the obvious place to introduce literals into the data path or to return values from the data path to the controller.

At the next higher level of system architecture, the shift array bit slice may be viewed as a system element with horizontal paths consisting of the bus, the shifter output, and if necessary, the shift constant since it appears at both edges of the array. The literal port is available into or out of the top edge of the bit slice, and the shift constant is available at the bottom of the bit slice. These slices, of course, are stacked to form a shift array as wide as the word of the machine being built.

One more observation concerning the multibit shifter is in order. We stated earlier that our data path was to have two buses. Therefore, in our data path, any bit slice of a shifter such as the one shown in Fig. 5.15 will of necessity have two buses running through it rather than one. We chose to show only one for the sake of simplicity. There remains the question of how the two buses are to be integrated with the shifter. Since we are constructing a two-bus data path, we have two full words available, and a good field extraction shifter would allow us to gracefully extract a word that crosses the boundary between two data path words. The arrangement shown in Fig. 5.15 performs a barrel shift on the word formed by one bus. Using the same number of control lines and pass transistors, and adding only the bus lines that are required for the balance of the data path anyway, we may construct a shifter that places the words formed by the two buses end to end and extracts a full-width word that is continuous across the word boundary between

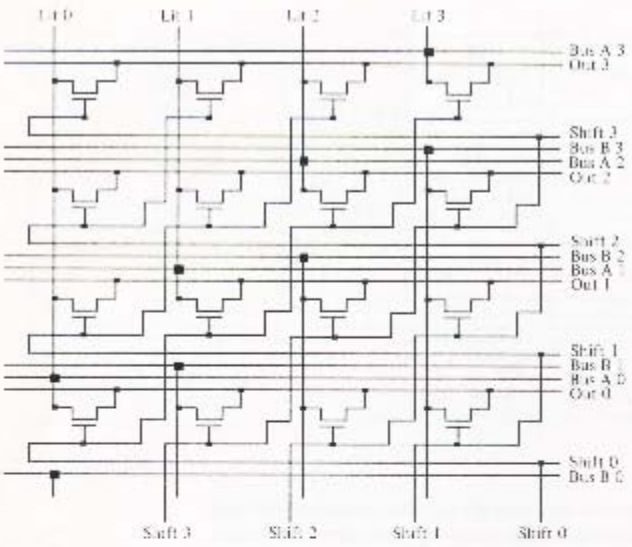



**Fig. 5.16** A 4-by-4 shifter with split vertical wires and two data buses.

the A and B buses. This function is accomplished, in as compact a form as just described, with a circuit shown in Fig. 5.16. Notice that the vertical wires have a split in them. The portion of the wire above the corresponding shift output is connected to bus A, and that below the corresponding shift output to bus B. The layout of the barrel shifter is shown in color Plate 13.

It can be seen by inspection that this circuit performs the function shown in Fig. 5.17, which is just what is required for doing field extractions and variable word length manipulations. The literal port is connected directly to bus A and may be run backward in order to discharge the bus when a literal is brought in from the control port. A block diagram representing the shifter at the next level of abstraction is shown in Fig. 5.18.

In order to complete the shifter functional block, it is necessary to define the drivers on the top and bottom that interface with the system at the next higher level. Let us assume that the literal bus from outside the chip will contain data valid on the opposite phase of the clock from that of the internal buses. For that case, a very simple interface between the two buses that will operate in either direction is shown in Fig. 5.19.

The internal shifter output is precharged during $\varphi_2$, and active during $\varphi_1$. It may be sourced from the shifted combination of either the A and B buses or the literal bus and B bus, as shown in Fig. 5.17. The external literal bus itself may be

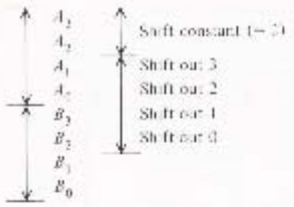



Fig. 5.17 Conceptual picture of the shifter's operation.

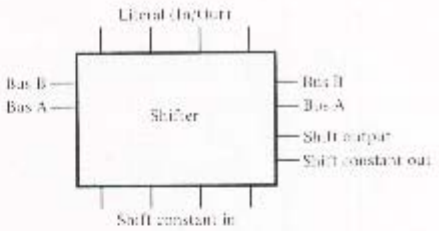



Fig. 5.18 Block diagram of the shifter.

sourced either from the opposite end (the external paths from the program source) or from the end attached to bus A in the shift array shown.

The bus to the external literal path is precharged during $\varphi_1$, and data bits from the literal port of the shifter are enabled onto it by a signal active during $\varphi_2$, as shown in Fig. 5.19. The two signals, $\varphi_1 *$ In, and $\varphi_2 *$ Out, are derived from buffers identical to those shown earlier. The shift constant itself is represented by one line out of $n$, which is high, the others remaining low. Buffers for these lines are identical to those shown in Fig. 5.10.

There is one more observation concerning the $n$-bit shift constant. It is represented most compactly by a log $n$ bit binary number. However, in order to generate from such a form a signal that can be used in the actual data path, a decoder is required for converting the binary number into a 1-of-$n$ signal suitable for feeding the buffers. There are a number of ways of making decoders in $n$MOS technology. The most common form is the NOR form, which is the fully decoded equivalent of the AND-plane in the programmable logic array (Chapter 3). It is shown in Fig. 5.20. Notice that the output is a high-going 1-of-$n$ pattern.

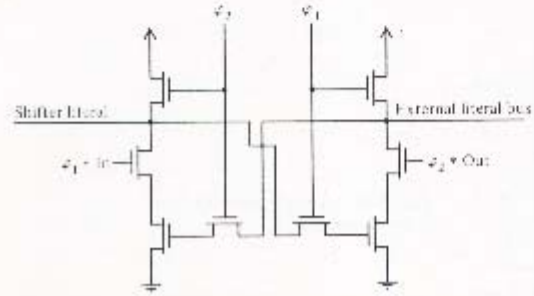



Fig. 5.19 Literal interface.

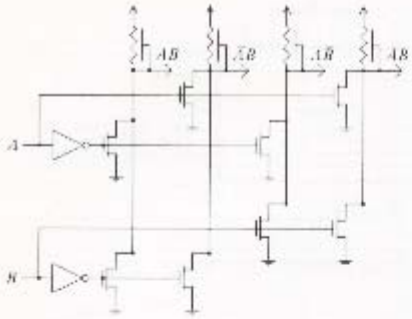


Fig. 5.20 A NOR form 1-of-$n$ decoder.

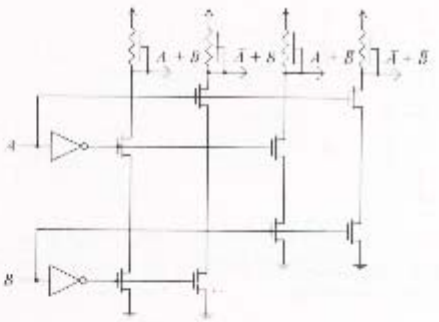


Fig. 5.21 A NAND form 1-of-$n$ decoder.

Decoders can also be made in other forms. For small values of $n$, the NAND form shown in Fig. 5.21 is often convenient. We used a variant of this form for the ALU function block described earlier. Notice that the output of this form, when used as a decoder, is a low-going 1-of-$n$ pattern. There is also a complementary form of decoder that can be built with this technology, as suggested by Ivan Sutherland. It takes advantage of the fact that in any decoder both the input term and its complement must be present. In this case, the input term can be used to activate pull-up transistors in series, while the complement can be used to activate pull-down transistors in parallel. This logic form is similar in principle to that used with complementary technologies and has similar benefits. It can generate either a high-going or a low-going 1-of-$n$ number and dissipates no static power. A decoder of this sort is shown in Fig. 5.22. Once we have added the appropriate buffers and decoders to our shift array, we have a fully synchronized subsystem ready to be integrated with the system at the next level up. The properties of this subsystem are shown in Fig. 5.23.

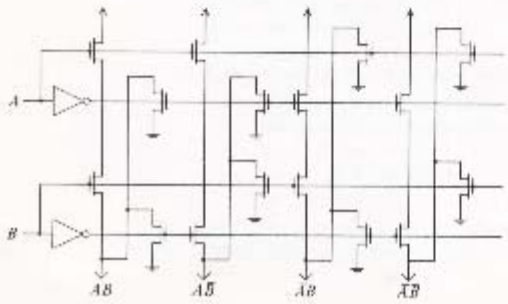


Fig. 5.22 A complementary form 1-of-$n$ decoder.

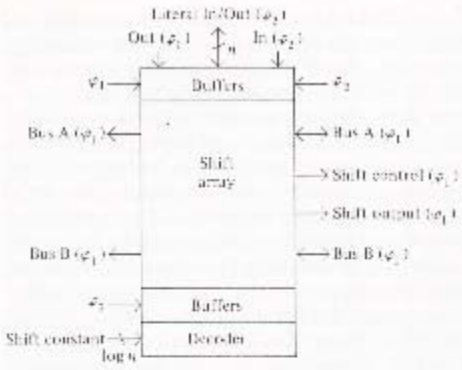



Fig. 5.23 A fully synchronized shifter.

### 5.9 REGISTER ARRAY

In any microprogrammed processor designed for emulating a higher level instruction set, it is convenient to have a number of miscellaneous registers available, both for working storage during computations and for storing pointers (stack pointers, base registers, program counters, etc.) of specific significance in the machine being emulated. Since the data path has two buses and the ALU is a two-operand subsystem, it is convenient if the registers in the data path are two-port registers. The circuit design of a typical two-port register cell is shown in Fig. 5.24. The layout of a pair of these cells is shown in color Plate 14. This register is a simple combination of the input multiplexer described earlier, the $\varphi_2$ feedback transistor, and two output drivers, one for each bus. The registers can be combined into an array $m$ bits long and $n$ bits wide. Each cell of the array can be viewed at the next level up as shown in Fig. 5.25. Drivers for the load inputs and the read outputs are

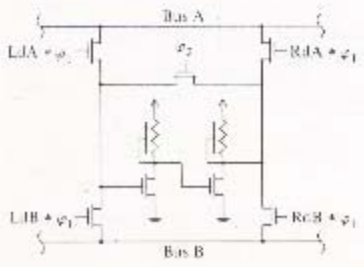



Fig. 5.24 A two-port register cell.

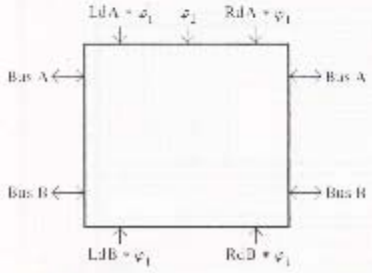



Fig. 5.25 Block diagram definition of the two-port register cell.

identical to those shown in Fig. 5.10. While we could immediately encode the load and read inputs to the registers into log $n$ bits, we shall delay doing so until the next level of system design. There are a number of sources for each bus besides the registers, and we will conserve microcode bits by encoding them together.

Before we proceed, there is one important matter that must be taken care of in the overall topological strategy. Routing of VDD and ground paths must generally be done in metal, except for the very last runs within the cells themselves. Often the metal must be quite wide, since metal migration tends to shorten the life of conductors if they operate at current densities much in excess of 1 milliampere per square micron cross section. Thus, it is important to have a strategy for routing ground and VDD to all the cells in the chip before doing the detailed layout of any of the major subsystems. Otherwise, one is apt to be faced with topological impossibilities because certain conductors placed for other reasons interfere with the routing of VDD and ground. A possible strategy for the overall routing of VDD and ground paths is shown in Fig. 5.26. Notice that the VDD and ground paths form a set of interdigitated combs, so that both conductors can be run to any cell in the chip. Any such strategy will do, but it must be consistent, thoroughly thought through at the beginning, and rigidly adhered to during the execution of the design.

## 5.10 COMMUNICATION WITH THE OUTSIDE WORLD

Although in particular applications the interface from a port of the data path to the outside world may be a point-to-point communication, the ports will often connect

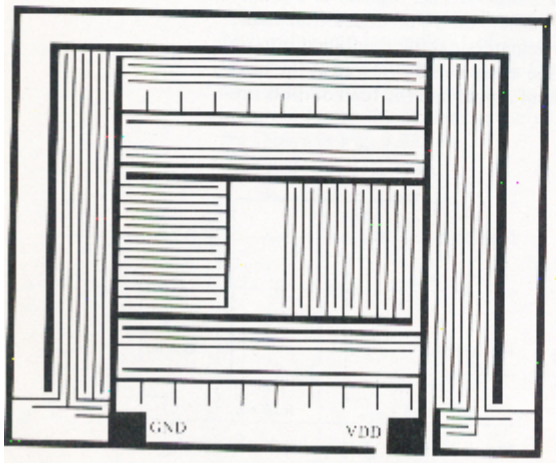



Fig. 5.26 VDD and GND net for the data path chip.

to a bus. Thus it is desirable to use port drivers that may be set in a high impedance state. Drivers that can either drive the output high, drive the output low, or appear as a high impedance to the output are known as *tri-state* drivers. Such drivers allow as many potential senders on the bus as necessary. Figure 5.27 shows the circuit for a tri-state interface to a bonding pad. Here, either bus A or bus B can be latched into the input of a tri-state driver during $\varphi_1$. Likewise the pad may be latched into an incoming register at any time independent of the clocking of the chip. Standard bus drivers are enabled on bus A and bus B. The only remaining chore is the design of the tri-state driver that drives the pad directly. Details of the tri-state driver are shown in Fig. 5.28. The layout of an output pad and its associated driver circuitry is shown in color Plate 15.

The terms *out* and *outbar* are fed to a series of buffer stages that provide both true and complement signals as their outputs and are disabled by a *disable* signal. Note that this Disable signal does not cause all current to cease flowing in the drivers, since the pull-up transistors are depletion type. In general there will be a number of super buffer stages of this sort. The very last stage of the driver is shown in Fig. 5.28(b). It is not a super buffer but employs enhancement mode transistors for both pull-up and pull-down. These transistors are very large in order to drive the large external capacitance associated with the wiring attached to the pad. They are disabled in the same manner as the super buffers, except that when the gates of both transistors are low, the output pad is truly tri-stated. The

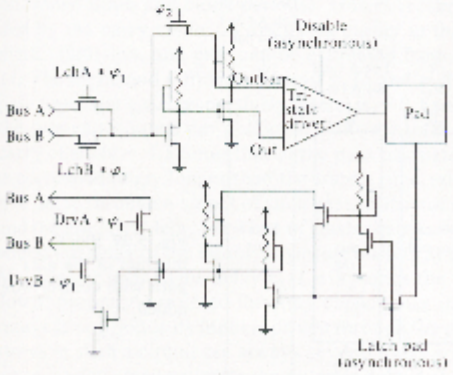



Fig. 5.27 Data port tri-state pad circuit.

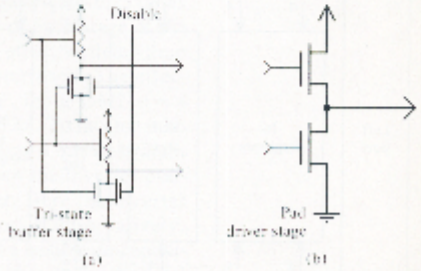



Fig. 5.28 The tri-state driver consists of any number of tri-state buffer stages followed by a pad driver stage. The current design used two tri-state buffer stages.

two output transistors are a factor of approximately $e$ larger than the last super buffer in the buffer string.

As we have seen, a rather large inverter string is required to transform the impedance from that of the internal circuits on chip to that sufficient for driving a pad attached to wiring in the outside world; the large size imposes a delay, of some factor times a logarithm of this impedance ratio, upon communications between the chip and the outside world. Any help that can be obtained in making this transformation is of great value. For example, the latch and buffers associated with the input bus circuit to the pad drivers can themselves be graded in impedance level, so that by the time the out and outbar signals are derived, they are at a considerably higher current drive capability than the buses. Note that the buses are of a considerably larger capacitance than minimum nodes on the chip, and thus the initial latch buffers can be larger than typical inverters on the chip. All such tricks help to minimize the number of stages between the bus and the outside pad and consequently the total delay in going off-chip.

### 5.11 ENCODING THE CONTROL OPERATION OF THE DATA PATH

By now we have defined a complete functional data path with ports on each end and functional blocks through the center, as shown in Fig. 5.29. The data path operation code bits required to control the data path are shown, as is the phase of the clock on which they are latched. There are forty-nine such bits together with

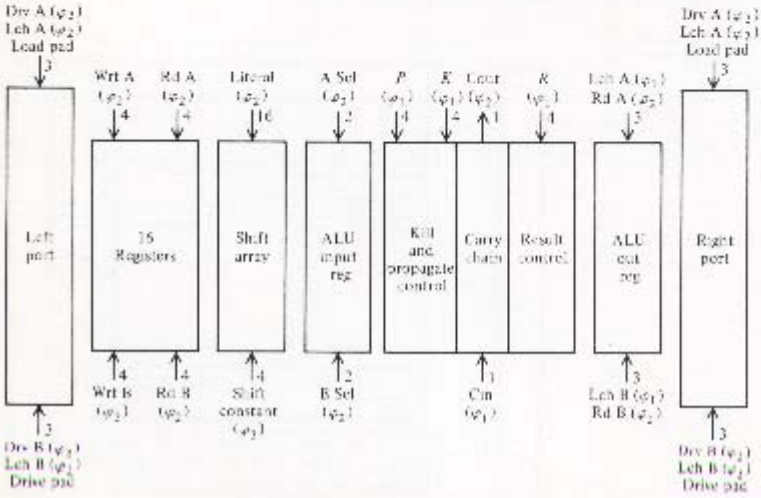



**Fig. 5.29** Block diagram of the data path with control wires added.

the four asynchronous bits for latching and driving the pad to the external world. In addition, there are the carry-out wire and the sixteen literal wires. These sixty-six wires together with the thirty-two from the left and right ports must go to and come from somewhere. (Schemes for encoding internal data path operations into microinstructions of various lengths are discussed in Chapter 6.) At one extreme all the data path control wires can be brought out to a microcode memory driven by a microprogram counter and controller, in which case all operations implementable by the data path may be done in parallel. The opposite extreme is to tightly encode the operations of the data path into a predefined microinstruction set. In the present system, this encoding would be most conveniently done by placing a programmable logic array or set of programmable logic arrays along the top and the bottom of the data path. A condensed microinstruction could then be fed to the programmable logic arrays that would then decode the compact microinstruction into the data path operation code bits.

An important point of the design strategy used here is that we can orthogonalize the design of the data path and the design of the microinstruction set in such a way that the interface between the two designs is not only very well defined and very clean, but it can be described precisely, in a way that system designers at the next higher level can understand and work with comfortably. The data path can then be viewed as a component in the next level system design.

Using the approximate capacitance values given in Table 2.1, we can estimate the minimum clock period for sequencing the data path. We would expect a $\varphi_1$ time for the data path of $\simeq 50\,\tau$ (same as the general estimate given in Section 1.13 on transit times and clock periods). However, the $\varphi_2$ time of the data path is limited by the carry chain, as discussed earlier in this chapter. The relative areas of metal, diffusion, and gate can be estimated from the ALU layout shown in Plate 11. The metal and diffusion occupy $\simeq$ 15 and $\simeq$ 8 times the area of the propagate pass transistor gate, respectively. Metal is $\simeq$ 0.1 and diffusion is typically 0.25 times the gate capacitance per unit area. Thus the total capacitance of each stage of the carry chain is $\simeq$ 4.5 times that of the pass transistor gate. The effective delay time is correspondingly longer than the transit time $\tau$ of the transistor itself. The effective delay through $n$ stages of such pass transistor logic is $\simeq \tau n^2$. In the OM2, $n = 4$ and the effective delay for 4 bits of carry chain is $\simeq 4.5 * 16\tau = 72\tau$. To this must be added the delay of the doubly inverting buffers at the end of every 4 bits of straight Manchester logic. This delay is $(1 + k)$ times the transit time of the inverter pulldown, properly corrected for stray capacitance in the inverter. Here the inverter ratio $k$ is $\simeq$ 8, since its input is driven through the pass transistors. Conservatively, strays in such a circuit are always several times greater than the basic gate capacitance, and we may estimate the inverter delays at $\simeq 30\,\tau$. Our estimate for the total carry time is thus $\simeq$100 times the transit time for each block of 4 ALU stages. The total $\varphi_2$ time should then be $\simeq 400\,\tau$. In 1978, the fastest commercial $n$ MOS processes yield a transit time $\tau$ of $\simeq$ 0.3 ns, and we would expect a minimum total clock period of $\simeq 450\,\tau$, or $\simeq$ 135 ns.

## 5.12 FUNCTIONAL SPECIFICATION OF THE OM2 DATA PATH CHIP

### 5.12.1 Introduction

This specification describes a 16-bit data path chip referred to as OM2 [#986]. The OM2 contains 16 registers, an ALU, and a 16-bit shifter, and is designed as part of a microprogrammed writable-control-store digital computer. The companion chip is the controller chip, which contains the program counter, stacks, and so on. The controller is described in Chapter 6. The entire system is designed to run on a single 5-volt supply.

The OM2 data chip has two data ports for communication with the external system and a communication path to the controller chip. The data ports are tri-state with either internal or external control. Communication with the controller consists of a 16-bit literal port and a single flag bit. Seven control bits come directly from the microcode memory.

The system runs on a single clock, generating $\varphi_1$ and $\varphi_2$ internally. When the clock is high, the internal buses transfer data; when the clock is low, the ALU is performing its operation. Microcode bits enter the data chip the phase before that code is to be executed. Therefore, the bus transfer code enters the data chip when the clock is low, and the ALU code enters when the clock is high. Figure 5.30 sketches a possible OM system.

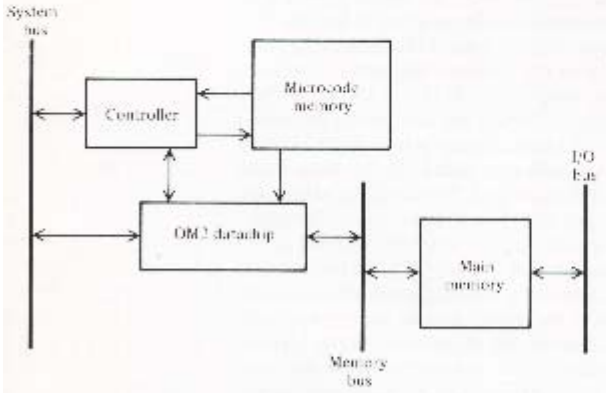



Fig. 5.30 One possible OM2 system configuration.

[1]Section 5.12 contains a functional specification of the OM2 data path chip, contributed by Dave Johannsen of California Institute of Technology. This specification was originally documented in Display File #1111, by Dave Johannsen and Carver Mead of the Caltech Computer Science Department, and copyrighted by Caltech. The specification is reprinted here with the permission of the California Institute of Technology. See also the later document, "Our Machine: A Microcoded LSI Processor," Display File #1826, by Dave Johannsen, Caltech Computer Science Department, for a general description of the OM System.

Throughout this section a positive logic convention is used. A "1" refers to a high voltage level, while a "0" refers to a low voltage level.

### 5.12.2 Data paths

A block diagram of OM2 is shown in Fig. 5.31. There are two buses that connect the various elements of the chip. The buses transfer data while the clock is high, the period referred to as $\varphi_1$. During $\varphi_2$, when the clock is low, the buses are precharged. Each bus can get data from only one source and give data to only one destination during any one cycle.

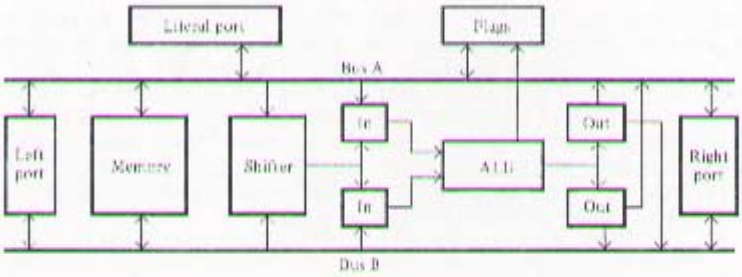



**Fig. 5.31** Block diagram of the OM2.

The left and right ports communicate between the data chip and the outside world. The right port has been traditionally known as the memory bus port while the left port has been the system bus port, but since the two ports are identical, this is an arbitrary convention. Each port has both an input latch and an output latch to provide facilities for synchronizing the data chip to the outside buses. Under program control either of the two buses can load the output latch during $\varphi_1$. There are three modes of driving data from the output latch to the pins, two of which are under program control and one of which is under hardware control. The first method is to output the data as soon as it comes from the bus, during the same $\varphi_1$. The second method is to latch the data from the bus during $\varphi_1$ and drive it out during the following $\varphi_2$. The final method is to latch the data from the bus during $\varphi_1$, but output the data when an enable pin is pulled low. The enable pin would be controlled by a bus manager, and can be asynchronous with respect to the data chip. Inputting from the port is similar. By pulling down on another enable pin, data bits from the external bus are loaded into the input latch, which can be read later under program control. Alternatively, the microcode can force the data currently on the external bus into the internal bus during the current $\varphi_1$. With this scheme, many types of synchronous and asynchronous buses may be interfaced to OM2's. For internal control only, the external enable pins can be left floating.

### 5.12.3 Registers

The registers are static and dual port. Any one of the 16 registers may source either or both of the buses, while any one of the 16 may be the destination for either bus, but not both. There are only two restrictions on the use of the registers:

1. One register may not be the destination for both buses on the same cycle.
2. One register may not be both the source for one bus and the destination for the other bus on the same cycle.

### 5.12.4 Shifter

The shifter concatenates the two buses, resulting in a 32-bit word, with bus A being the more significant half. The shift constant then selects the bit position where the 16-bit output window starts. The shift constant specifies the number of bits from bus B present in the output; i.e., a shift constant of 0 returns bus A, while a shift constant of 15 returns the LSB of bus A in the MSB of the output, followed by all but the LSB of bus B in the rest of the word. A conceptual picture of the shifter is shown in Fig. 5.32. The ALU can select as inputs either the bus, the shift output, or shift control. If shift control is selected, the entire word is 0 except where the LSB of bus A appears in the shift output. The shifter operates on $\varphi_1$; it may be viewed as an extension of the buses.

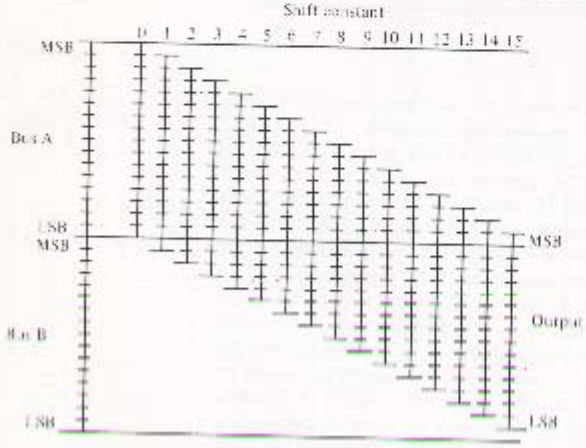



Fig. 5.32 Shifter operation.

### 5.12.5 ALU

A block diagram of a single bit of the ALU is shown in Fig. 5.33. The ALU operates on the data that is contained in its two input latches. Input latch *A* may be

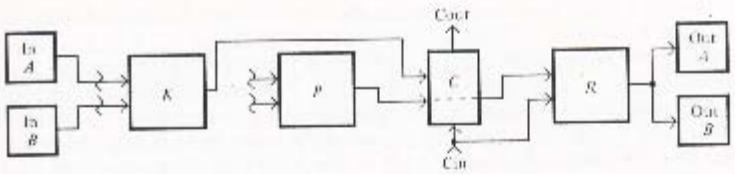


Fig. 5.33 Block diagram of one bit of the ALU.

loaded from bus A, the shifter output, or the shift control, while the input latch *B* may be loaded from bus B, the shifter output, or the shift control.

The outputs of the two latches become the inputs to two function blocks that determine what will happen on the carry chain. Function block *P* determines whether the carry chain propagates, while *K* decides if it is to kill the carry. If neither is true, the carry chain generates a carry. Each function block has four control inputs, which, for the propagate function block, are referred to as *PFF*, *PFT*, *PTF*, and *PTT*. If *PFF* is enabled, the *P* block output is high if both input latches are false (contain 0). Enabling *PFT* activates the output if input *A* is false and input *B* is true, and so on. If, for example, both *PFF* and *PFT* are enabled, the output is active if input *A* is false, regardless of the state of input *B*. To further illustrate the operation of the function blocks, consider addition. If both inputs contain a 1, the carry is to be generated, while if both inputs are 0, the carry is killed. If the two inputs are different, the carry is to be propagated (carry-out = carry-in). To do this operation, the kill output should be active if both inputs are false, so *KFF* is enabled. Both *PFT* and *PTF* should be enabled to propagate properly. Therefore, $K = (KFT, KFT, KTF, KTT) = (1,0,0,0)$, and $P = (PFF, PFT, PTF, PTT) = (0,1,1,0)$.

The result of the ALU is produced by the *R* function block, which has as inputs *P*-block out and carry-in. For the addition example above, the output should be the exclusive-OR of *P* and Cin, so $R = (0,1,1,0)$. *P*, *K*, and *R* values for common ALU operations are listed in Table 5.2.

Two ALU output latches (*A* and *B*) can be loaded from the *R* block output; either one may later be used to source either bus.

### 5.12.6 Flags

The carry input to the LSB of the ALU is a logical combination of a flag bit and two control inputs. The two control inputs can force the carry-in to be either 1 or 0, or they can select either flag or flag bar as the input.

There is also a method for doing conditional ALU operations under the control of a 2-bit conditional OP field. A conditional operation performed by the ALU is not only a function of the control inputs but also of the flag bit. The conditional operation control forces some of the control inputs low, regardless of what the *P*, *K*, and *R* microcode says. The coding for conditional operations allows the use of operations like multiply step and divide step without the necessity for branching in the microcode.

There is a 16-bit flag register that can also be a source or destination of bus A. This register can also be loaded with the ALU flags during $\varphi_2$. The ALU flags include *carry-out, overflow, carry-in to the MSB, zero, MSB, LSB, less than, less than or equal to,* and *higher* (in unsigned value). The last three flags are comparison flags used after a subtraction. For example, after subtracting ALU input latch *B* from latch *A*, the "less than" flag is true if the value in ALU input latch *B* is larger than the value in ALU input latch *A*. The MSB of the flag register is called the flag bit, and this bit may be modified every $\varphi_1$ by loading it with the value of one of the other bits of the flag register. The flag bit is used in the calculation of carry-in and modification of conditional ALU OPs. This bit is also sent to the controller chip to be used for conditional branching, etc.

### 5.12.7 Literal

The one remaining data path is the literal port. It is used to send data from the data chip to the controller, and vice versa. It is a source or destination for bus A. When the literal port is being used, standard bus operations are suspended for that cycle.

### 5.12.8 Programming

The data chip requires 23 bits of microcode on each phase of the clock. This section of the memo specifies the encoding of the fields within that microcode. Figure 5.34 shows the arrangement of the microcode word.

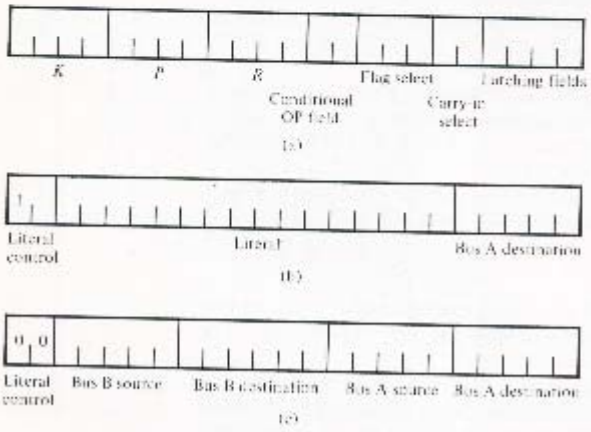



**Figure 5.34**
(a) Phase 2 OP code (in on $\varphi_1$).

(b) Phase 1 literal transfer OP code (in on $\varphi_2$).

(c) Phase 1 normal OP code (in on $\varphi_2$).

***Bus Transfer***

The bus transfer control bits enter the data chip during $\varphi_2$ and are used during the following $\varphi_1$. There are two buses, bus A and bus B, that interconnect the modules

Table 5.1

| Bus A Source | | Bus A Destination | |
|---|---|---|---|
| 0*nnnn* | Register *n* | 0*nnnn* | Register *n* |
| 10000 | Right port pins | 10000 | Left port, drive now |
| 10001 | Right port latch | 10001 | Left port, drive $\varphi_2$ |
| 10010 | Left port pins | 1001x | Left port, no drive |
| 10011 | Left port latch | 10100 | Right port, drive now |
| 10100 | ALU output latch *A* | 10101 | Right port, drive $\varphi_2$ |
| 10101 | ALU output latch *B* | 1011x | Right port, no drive |
| 10110 | Flag register | 11000 | ALU input latch *A* |
| | | 11001 | ALU input latch *A* gets shift out |
| | | 11010 | ALU input latch *A* gets shift control |
| | | 11011 | Flag register |

| Bus B Source | | Bus B Destination | |
|---|---|---|---|
| 0*nnnn* | Register *n* | 0*Dnnnn* | Register *n* |
| 10000 | Right port pins | 010000 | Left port, drive now |
| 10001 | Right port latch | 010001 | Left port, drive $\varphi_2$ |
| 10010 | Left port pins | 01001x | Left port, no drive |
| 10011 | Left port latch | 010100 | Right port, drive now |
| 10100 | ALU output latch *A* | 010101 | Right port, drive $\varphi_2$ |
| 10101 | ALU output latch *B* | 0101Lx | Right port, no drive |
| | | 0110xx | ALU input latch *B* |
| | | 1D*nnnn* | ALU input latch *B* gets shift output, shift constant = *n* |
| | | 11*nnnn* | ALU input latch *B* gets shift control, shift constant = *n* |

of the data chip. These two buses are similar in many respects; however, there are a few asymmetries as to sources and destinations. Also, when a literal is being transferred, the only bus transfer field that is active is the bus A destination, which stores the literal entered on bus A. A listing of bus sources and destinations appears in Table 5.1.

### *ALU Input Selection*

The two ALU input latches are destinations for the two buses, as shown above. In addition to being loaded directly from the buses, these two latches can be loaded from the outputs of the shift array. The shift constant always comes from the four least significant bits of the bus B destination field, even though the destination of bus B is not the ALU input latch *B*. For example, bus B may be transferring the contents of register 3 into register 5 while bus A is transferring the contents of register 4 to the ALU input latch *A* through the shifter. In this case, the shift constant would be "5" because the four least significant bits of the bus B destination field contain "0101".

Table 5.2

| | K | P | R | Cin | Cond | |
|---|---|---|---|---|---|---|
| $A + B$ | 1 | 6 | 6 | 0 | 0 | Add |
| $A + B + \text{Cin}$ | 1 | 6 | 6 | 1 | 0 | Add with carry |
| $A - B$ | 2 | 9 | 6 | 2 | 0 | Subtract |
| $B - A$ | 4 | 9 | 6 | 2 | 0 | Subtract reverse |
| $A - B - \text{Cin}$ | 2 | 9 | 6 | 1 | 0 | Subtract with borrow |
| $B - A - \text{Cin}$ | 4 | 9 | 6 | 1 | 0 | Subtract reverse with borrow |
| $-A$ | 12 | 3 | 6 | 2 | 0 | Negative $A$ |
| $-B$ | 10 | 5 | 6 | 2 | 0 | Negative $B$ |
| $A + 1$ | 3 | 12 | 6 | 2 | 0 | Increment $A$ |
| $B + 1$ | 5 | 10 | 6 | 2 | 0 | Increment $B$ |
| $A - 1$ | 12 | 3 | 9 | 2 | 0 | Decrement $A$ |
| $B - 1$ | 10 | 5 | 9 | 2 | 0 | Decrement $B$ |
| $A \wedge B$ | 0 | 8 | 12 | 0 | 0 | Logical AND |
| $A \vee B$ | 0 | 14 | 12 | 0 | 0 | Logical OR |
| $A \oplus B$ | 0 | 6 | 12 | 0 | 0 | Logical EXOR |
| $\neg A$ | 0 | 3 | 12 | 0 | 0 | Not $A$ |
| $\neg B$ | 0 | 5 | 12 | 0 | 0 | Not $B$ |
| $A$ | 0 | 12 | 12 | 0 | 0 | $A$ |
| $B$ | 0 | 10 | 12 | 0 | 0 | $B$ |
| Mul | 1 | 14 | 14 | 0 | 1 | Multiply step |
| Div | 3 | 15 | 15 | 0 | 2 | Divide step |
| $A \vee O$ | 0 | 14 | 12 | 0 | 3 | Conditional AND/OR |
| Mask | 10 | 5 | 8 | 2 | 0 | Generate mask |
| SHL A | 3 | 0 | 10 | 0 | 0 | Shift A left |
| Zero | 0 | 0 | 0 | 0 | 0 | Zero |

### *ALU Operations*

Table 5.2 shows coding for ALU operations that are commonly found useful. The user is encouraged to encode other operations if these are not suitable. The numbers given are the decimal representation of the 4-bit control word. For $P$ and $K$, $A'B' = 1$, $A'B = 2$, $AB' = 4$, $AB = 8$. For $R$, $P'C' = 1$, $P'C = 2$, $PC' = 4$, $PC = 8$. Cin is the carry-in select, and Cond is the conditional OP select.

### *Carry-In Select*

The carry-in select field determines what the carry into the LSB of the ALU will be, according to the following table:

| | |
|---|---|
| 00 | 0 |
| 01 | Flag bit |
| 10 | 1 |
| 11 | Flag bit complemented |

***Conditional OP Select***

The conditional OP select field is used to generate three basic conditional type operations: multiply, divide, AND/OR step. In a great many cases, the conditional OP allows functions dependent on a flag to be performed in one cycle, rather than sending the flag to the controller and branching to two separate instructions depending upon that flag. When a conditional OP is selected, certain ALU control bits are forced to zero. Which bits are zeroed depends on the conditional OP select and the flag bit, as follows:

| Select | Flag bit | $K$ | $P$ | $R$ | |
|---|---|---|---|---|---|
| 0 | x | ---- | ---- | ---- | Unconditional |
| 1 | 0 | ---0 | --0- | --0- | Multiply step |
| | 1 | ---- | 0--- | 0--- | |
| 2 | 0 | 0--0 | -00- | -00- | Divide step |
| | 1 | -00- | 0--0 | 0--0 | |
| 3 | 0 | ---- | ---- | ---- | AND/OR |
| | 1 | ---- | -00- | ---- | |

For example, consider multiplication: If the flag bit is high, $P_3$ and $R_3$ are grounded, so the ALU OP (1,14,14) becomes (1,6,6), which is the code for "ADD". If the flag bit is low, $K_0$, $P_1$, and $R_1$ are pulled low, transforming (1,14,14) into (0,12,12), i.e., the code for "input A".

***Flags***

The flag select field determines which of the ALU flags becomes the new flag bit. The following table lists the selection options.

| Select | New flag bit |
|---|---|
| 0 | Old flag big |
| 1 | Carry-out |
| 2 | MSB |
| 3 | Zero |
| 4 | Less than |
| 5 | Less than or equal |
| 6 | Higher (in absolute value) |
| 7 | Overflow |

The ALU flags are loaded into the flag register under the control of the latching field, bit 3. They are loaded into the following positions.

| Bit | Flag |
|---|---|
| 0 | Not changed |
| 1 | Not changed |
| 2 | Not changed |
| 3 | Not changed |
| 4 | Not changed |
| 5 | Previous value of flag bit |
| 6 | ¬ Carry into MSB stage |
| 7 | Less than or equal |
| 8 | — Higher (in absolute value) |
| 9 | ¬ Less than |
| 10 | LSB |
| 11 | ¬ Zero |
| 12 | MSB |
| 13 | Overflow |
| 14 | ¬ Carry out |
| 15 | Current flag bit |

***Latching Field***

The latching field specifies which of four registers should be loaded, as shown in the following table:

| Latching field | Register loaded |
|---|---|
| 1xxx | Flag register loaded with current ALU flags |
| x1xx | ALU output latch *A* loaded with the ALU output |
| xx1x | ALU output latch *B* loaded with the ALU output |
| xxx1 | The literal field during the next $\phi_2$ is loaded with the contents of bus A during the last $\phi_2$ |
| 0000 | None of these registers are affected |

***Literals***

The 2-bit literal field specifies when a literal is to be used and which direction it goes. If both bits are 0, no literal transaction will occur. If the first bit is 1, a literal will be transferred. If the second bit is 1, the literal goes off-chip, while if the bit is 0, the literal comes on-chip.

### 5.12.9 Programming Examples

Here we present three programming examples that should provide a better understanding of the various data paths within OM2.

The first example is 16-bit integer multiplication. The two inputs, $X$ and $Y$, are multiplied to produce the result, $Z$. In the multiply loop, the number $X$ is shifted left and the MSB is stripped off. $Z$ is shifted left, then $Y$ is added to the new $Z$ if the MSB of $Y$ was a 1. The sequence of instructions is repeated 16 times, using the counter in the controller to signal when the 16 iterations have been performed.

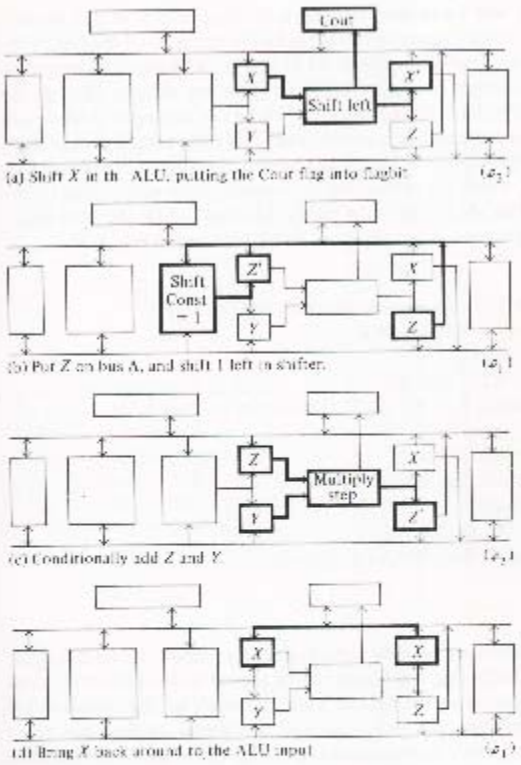



Figure 5.35

Assume contents of R[0] = 0. Figure 5.35 illustrates each step of the loop listed here:

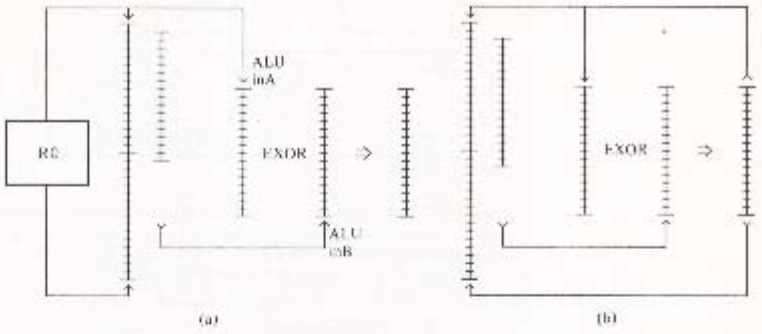
$\varphi_2$: ALU.Out.A←ALU(Shift A left)←ALU.In.A;
Latch Flags;
$\varphi_1$: ALU.In.A←Shift.out, Bus.A←ALU.Out.B;
R[1]← Bus.B← R[0]: This gives a shift constant of 1.
$\varphi_2$: ALU.Out.B←ALU(Multiply Step); *conditionally add.*
Flag←Cout;
$\varphi_1$: ALU.In.A←Bus.A←ALU.Out.A

The second example will be to generate a parity flag, which is not directly available from the ALU. Parity is generated by exclusive-oring all of the bits of the data together. If the data are loaded into both ALU inputs, with input $B$ rotated by one, performing an exclusive-or operation will give an output that is the exclusive-or of adjacent bits; bit $i$ of the output will be bit $i$ of the input $\oplus$ bit $i-1$ of the same input. If this same operation is performed, this time rotating input $B$ by two, bit $i$ becomes $i \oplus i-1 \oplus i-2 \oplus i-3$. By doing this two more times, rotating $B$ first by four and then by eight, every bit of the output is equal to the parity, that is, the EXOR of all of the bits. The MSB flag is the parity odd flag, while the zero flag is the parity even flag. The program is listed below and illustrated in Fig. 5.36.

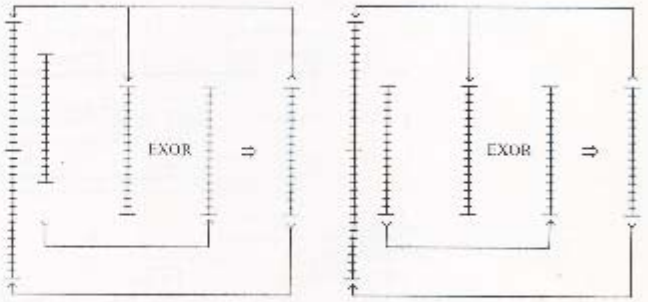
```
φ1: ALU.In.A←Bus.A←R[0];   generate the parity of register 0.
    ALU.In.B←Shift.out(1); Bus.B←R[0];
φ3: ALU.Out.A←ALU(Exor);
φ1: ALU.In.A←Bus.A←ALU.Out.A;
    ALU.In.B←Shift.out(2); Bus.B←ALU.Out.A;
φ2: ALU.Out.A←ALU(Exor);
φ1: ALU.In.A←Bus.A←ALU.Out.A;
    ALU.In.B←Shift.out(4); Bus.B←ALU.Out.A;
φ2: ALU.Out.A←ALU(Exor);
φ1: ALU.In.A←Bus.A←ALU.Out.A;
    ALU.In.B←Shift.out(8); Bus.B←ALU.Out.A;
φ3: ALU(Exor);
```

The third example illustrates how the data path can compute its own instruction. When driving a literal off-chip, the literal values appear in 16 of the microcode bits. If we have the literal port drive the data off chip, but don't set the disable bits in the instruction decoder, the data path will "execute" the literal. In the code below we sum all the registers (as further illustrated in Fig. 5.37). The basic literal transfers R[1] to the ALU to be added to R[0]: if we increment and execute the literal, then R[2] is transferred; etc.

```
φ1: ALU.In.A←Literal "Bus.A←R[1]; ALU.In.B←Bus.B←ALU.Out.B";
φ2: ALU.Out.B←ALU(A);
φ1: ALU.In.A←Bus.A←R[0];
φ2: ALU.Out.A←ALU(0);   This is just setup, now the loop!
φ1: Bus.A←ALU.Out.B;
    ALU.In.B←Bus.B←ALU.Out.A;
φ2: ALU.Out.A←ALU(add);
    Execute Literal;
φ1: ALU.In.A←A.Bus;   The rest of this instruction is the literal!
φ2: ALU.Out.B←ALU(increment B)←ALU.In.B;   point to next register.
```

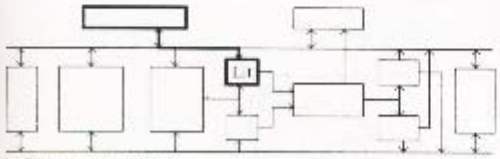



(a) (b)

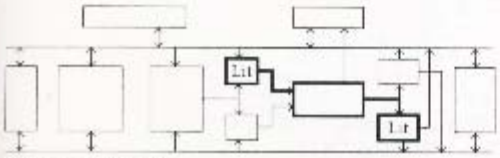



(c) (d)

**Fig. 5.36** (a) Shifting by 1: Result is exclusive-OR of adjacent bits. (b) Shifting by 2: Result is exclusive-OR of four adjacent bits. (c) Shifting by 4: Result is exclusive-OR of eight adjacent bits. (d) Shifting by 8: Result has all bits equal to the parity.



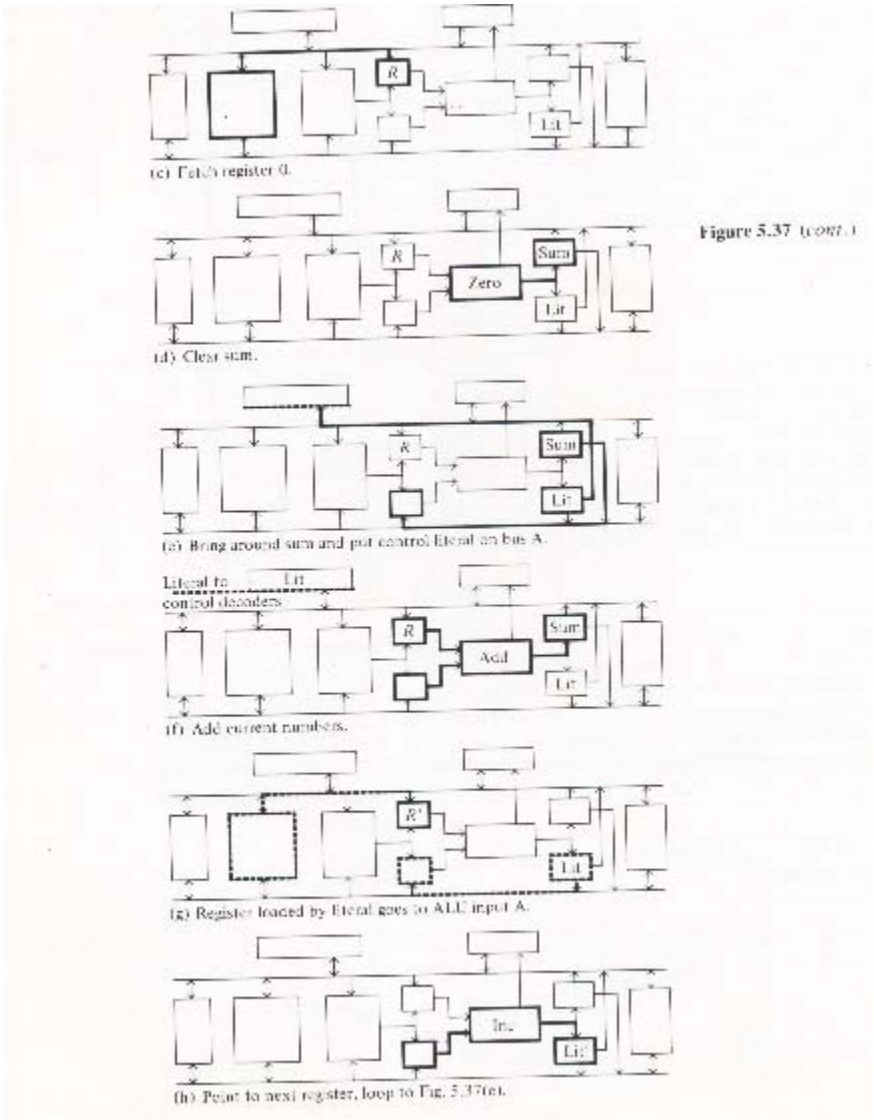

(a) Bring in control literal.




(b) Store in ALU.Out.lt.

**Figure 5.37**

*(Continued)*




(c) Fetch register 0.

(d) Clear sum.

(e) Bring around sum and put control literal on bus A.

(f) Add current numbers.

(g) Register loaded by literal goes to ALU input A.

(h) Point to next register, loop to Fig. 5.37(e).

Figure 5.37 (CONT.)