Chapter 5: Overview of an LSI Computer System, and the Design of the OM2 Data Path Chip

Copyright © 1978, C.Mead, L.Conway

Sections:

Up to this point, we have chosen simple examples to illustrate the fundamental properties of integrated systems, and have presented a design methodology which can be used to build hierarchically organized, complex systems. To more fully clarify these concepts, we now present examples drawn from the design of an LSI computer system. In this chapter, we provide a brief overview of this computer system, and then describe in detail one of its major components, the *data path chip*. Much of the detail in this chapter is intended to provide the reader with a source of examples of the implementation of digital logic subsystems into LSI circuit layout structures, under the constraints imposed both by the design methodology and by the architectural requirements of a real computer system. Chapter 6 similarly describes the *controller chip* of this computer system, and provides additional information on the sequencing of the overall system.

In this chapter we assume that the reader is familiar with the structure and function of the classical stored program digital computer, and with the concept, and computer design implications, of microprogrammed control. An informal review of these basic concepts is given in the introductory portions of chapter 6, so that the mapping of the required controller subsystems into silicon can be examined. The less experienced reader may benefit from a study of that material in parallel with reading this chapter.

It is important to note that the computer system discussed in chapters 5 and 6, while composed of structured LSI subsystems, is nevertheless of classical von Neumann form. The architectural possibilities of VLSI are just now beginning to be explored. Future lower cost, higher density, higher speed devices, combined with major reductions in integrated system implementation time, may make completely new forms of computing machines, and new notions of programming, not only feasible but also practical. Some of these issues will be discussed in chapter 8.

The OM Project at Caltech

The design of this computer system was undertaken as a university project in experimental computer architecture. The "Our Machine" (OM) project, as it has come to be known, was started by Carver Mead in 1976, as part of the LSI Systems course at Caltech. The project involves the design of a number of LSI chips, as described in the system overview section.

The initial focus of the project was the architecture and design of the system's primary data processing module, the *data path chip*. Early contributions to this effort were made by Mike Tolle [Litton Industries], while attending the LSI systems course. Other participants were Caltech students Dave Johannsen and Chris Carroll, with much inspiration from Ivan Sutherland. By December 1976, the first design (OM_0) of the data path chip was nearly completed. The participants decided at that time that the design had become "baroque" and ugly, and it was scrapped. A new data path design (OM1) was completed by March 1977 by Dave Johannsen, Chris Carroll, and Rod Masumoto. Fabricated chips were received in June 1977. It was this chip which appeared in the September 1977 Scientific American article by Sutherland and Mead. The chip was fully functional except for a timing bug in the dynamic register array, which had been designed in departure from the structured design methodology developed in this text.

A complete redesign of the data path chip was undertaken in June 1977, by Dave Johannsen. By September 1977, a complete set of new cells had been constructed. The design was completed by December, and chips fabricated by April 1978. The redesign included improvements in the encoding of the microcode control word, and rigorously applied the structured design methodology. Certain cells from the OM2 data path chip, and from its companion controller chip, were used as examples in chapter 3.

During 1977, the controller chip was designed as one of 4 class projects in the Caltech LSI systems course. It was finished in the summer of 1977, and fabricated chips were received in early 1978.

During 1978, the architecture of an overall system was planned. Design has begun of the three remaining chips in the OM computer system: the system bus interface chip, the memory manager chip, and the clock chip.

All of the detailed LSI design on the OM project has been done by students. Throughout most of the project's history only rather limited design aids were available, notably a simple symbolic

layout language and graphic plotters for checkplotting. The efforts of students to quickly create large integrated systems, using only primitive designs aids, helped to motivate the development and refinement of the structured design techniques described in this text.

The OM project has also required the implementation of many prototype designs and complete chip designs. Since early in the project, the Caltech group collaborated with researchers in industry, who were similarly completing many prototype LSI system designs, on the development of practical methods for simplifying and speeding up prototype project implementation. This led to the formulation and debugging of the standard starting frame for conveying multi-project chips through maskmaking and wafer fabrication, as described in chapter 4.

System Overview

An informal block diagram of one OM processor is shown in figure 1a. Such a processor 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 figure 1a. Several such processors may be interconnected via the system bus to augment one user's computer system. Tasks may then be distributed among the processors, improving system performance, for example by using different processors to independently control different input/output devices. Groups of different user systems may also share the system bus.

Each OM processor 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. For a more detailed description of these chips and the overall system, see reference 1.

The *data path chip* performs most of the data manipulation functions for the processor. These 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 file, a barrel shifter, and an arithmetic logic unit (ALU). Two buses connect these subsystems together. This chip's internal structure is described in detail later in this chapter.

The *controller chip* contains the microprogram counter (μPC) which stores the microcode memory address, and a counter for the control of microprogram loops. This 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 off the stack, and the manager does the rest, maintaining stack pointers, performing bounds checking to see if the stack is full or empty, etc.

The *system bus interface chip* provides asynchronous communication with other OM processors via the system bus. There are a whole host of subtleties associated with interfacing asynchronous buses. These issues are discussed in detail in chapter 7, along with the details of the organization and design of the interface chip.

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 a single $\varphi 1$ clock input, and generate the other clock phase signal on-chip.

A few words about timing may be helpful: In general, during $\varphi 1$ data is transferred from one subsystem to another on the same chip, while during $\varphi 2$ data is 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 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 during the study of the OM2 controller chip in chapter 6.

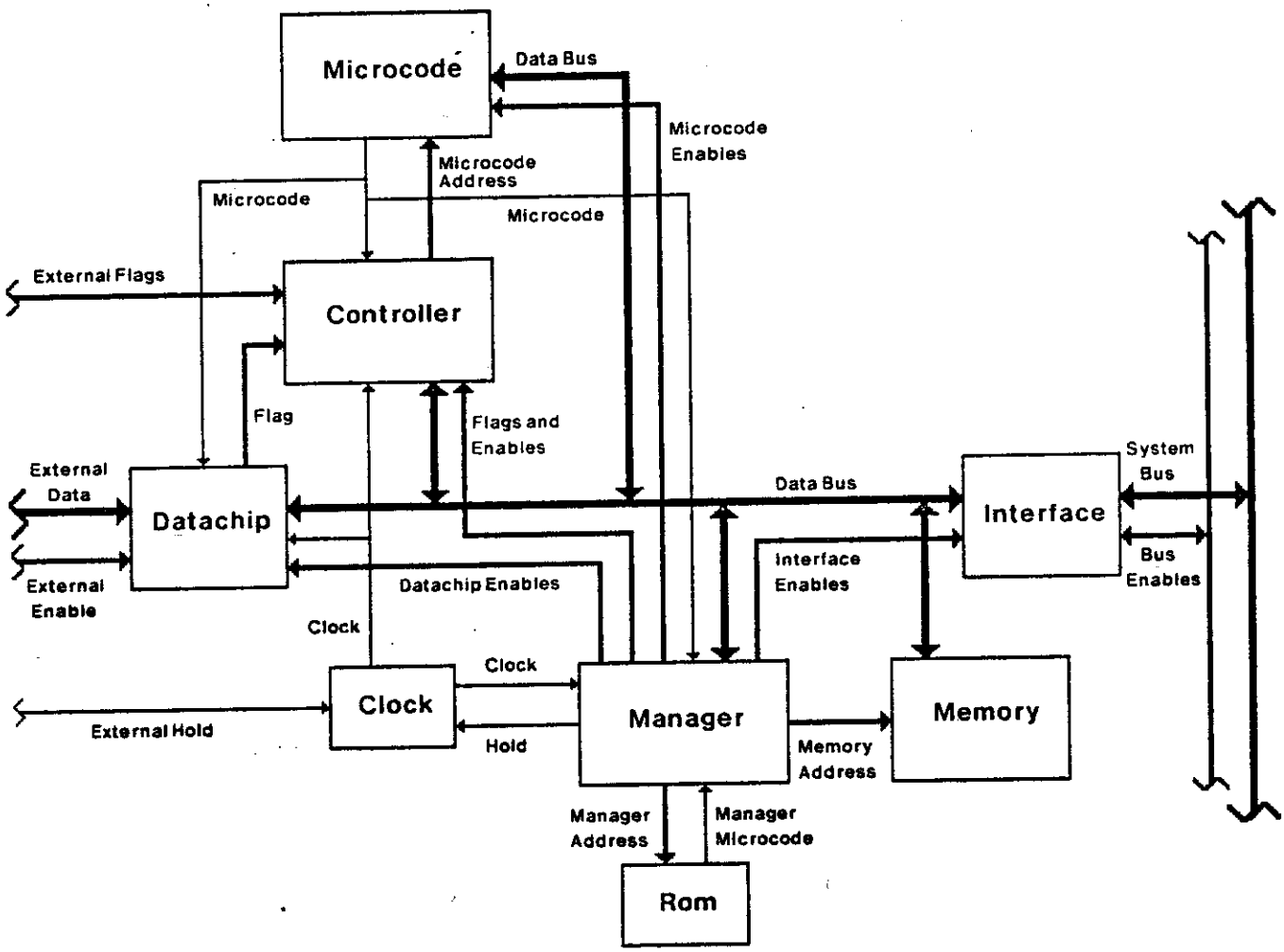


Figure 1a. Block Diagram of an OM Processor.

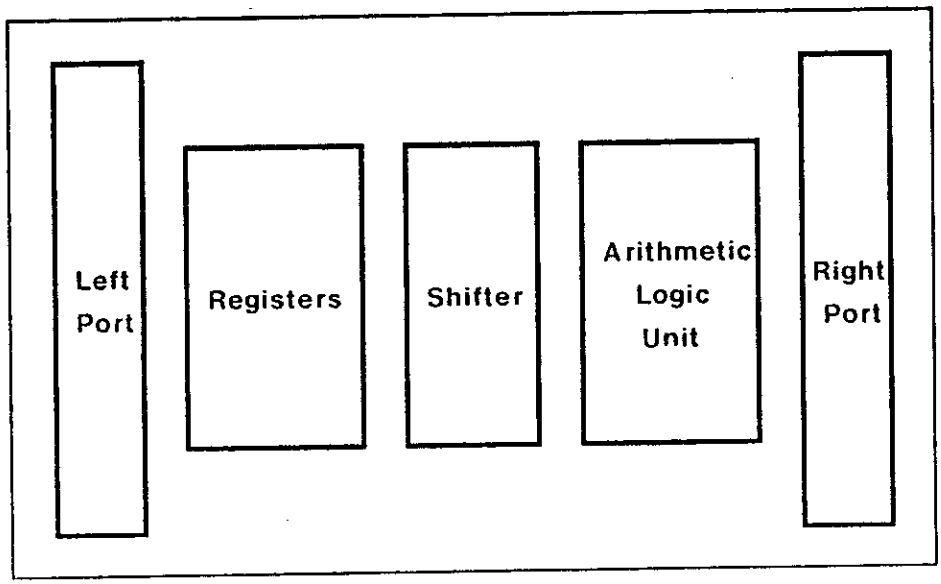


Figure 1b. General Floor Plan of the Data Path Chip.

The Overall Structure of the Data Path

The basic requirements initially established for the data path chip were (i) that it be gracefully interconnectable into multiprocessor configurations, (ii) that it effectively support a microprogrammed control structure, thus enabling machine instruction sets to be configured to the application at hand, (iii) that it be able to do variable field operations for emulation instruction decoding, assembly of bit-maps for graphics, etc., and (iv) 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. In many systems time is lost in assembling the two operands required for many operations. Therefore, the data path has two internal buses, and all registers on the chip are two-port registers. 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 order to avoid extensive random wiring for connecting the major subsystems on the chip, the following strategy was adopted at the outset: two system 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, and the two system buses were to run the full length of the chip between the two ports through the register and the data processing array.

The three main functional blocks on the chip are the register array, the shifter, and the arithmetic logic unit. 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 figure 1b. 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 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. Some of the layouts shown are earlier versions than those actually included in the final data chip. Nevertheless, they convey the basic ideas involved in laying out those cells. The overall layout of the data chip is shown in the frontispiece.

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 schemes indicated that they added 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 (reference 4, chapter 1), having a carry propogation circuit similar to that shown in figure 11, chapter 1. The carry chain and its associated logic were allowed to dictate the repeat distance of the cells in the vertical direction. In 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 any 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, and the resulting carry chain is shown in figure 2.

The main carry chain runs through the pass transistor from carry-in to carry-out. The carry-in signal is detected by the gate of an inverter which 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 which is derived from the inputs to the ALU, and simply grounds the carry-out through a single transistor. The third is a pass transistor which 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 which 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.3, the carry-chain may be seen as a logic block with 2 inputs, carry-kill' and carry-propagate', 2 outputs, propagate and carry-in, a vertical signal, carry-in and carry-out, and one control wire, precharge.

The task of designing the balance of the ALU is now reduced to that of designing functional

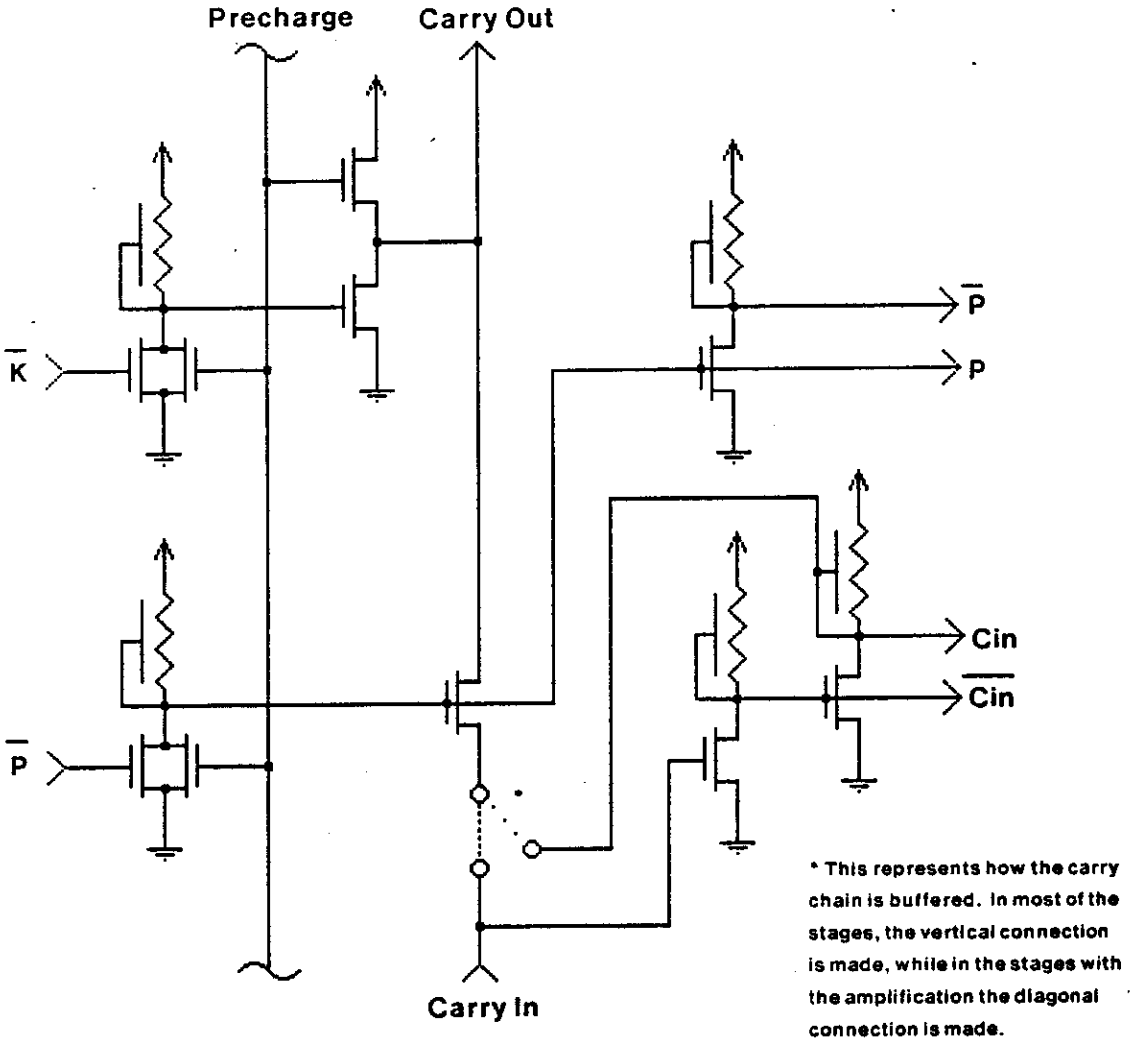


Figure 2. Carry Chain Circuit for the Arithmetic Logic Unit.

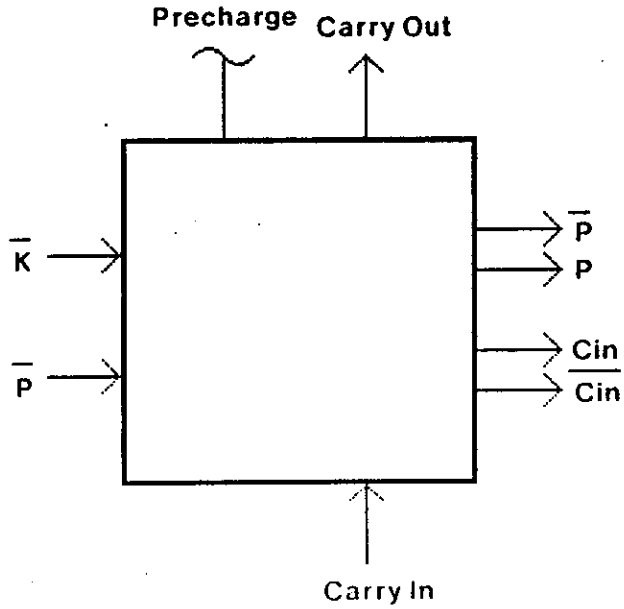


Figure 3. Abstraction of the Carry Chain Circuit.

blocks to: (a) combine the two input variables to form a propagate bar and kill bar, (b) combine carry bar and propagate to form the output signal, and (c) drivers for controlling the logical 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 logical function block illustrated in chapter 3, figure 12a. 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 carry-bar, propagate-bar, and for combining carry-bar in and propagate to form the output. The circuit, shown in figure 4, implements sixteen logic functions of two input variables. It consists of a set of transistors which fully decode the input combination of A and B, and connect 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 labelled 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 0 and 3, and logic-1 will be applied to control wires 1 and 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. 4 is shown in figure 5.

The block diagram for a complete arithmetic logic unit is shown in figure 6. 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 at a later time.

There are two general principles illustrated by this design. First, it is often less expensive in area, time, and power to implement a general function than to implement a specific one. Secondly, 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 choices of which functional entities to leave unbound and which to bind early requires a considerable amount of judgment, and is where much of the skill in integrated system design lies.

Two details need to be dealt with before the arithmetic logic unit function block is complete. Drivers are needed for the P_0 - - P_3, K_0 - - K_3, and R_0 - - R_3 control lines which 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 which 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-out signal from the pass transistor, we have doubly inverted our carry-in and buffered it to minimize the propagation delay. This approach avoids putting spaces between the carry function blocks for inverters. It is illustrated by the dotted connection lines in figure 2. In the actual implementation, the connection through the inverters was made in every fourth stage.

Drivers for the P, K, and R control lines have the following function: At some time during the null period of the ALU (which we shall call φ_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 buffer implementing this function is shown in Fig. 7.

The OP code is latched through a pass transistor whose gate is connected to φ_1, and the OP code runs into a NOR gate, the other input of which is also φ_1. Thus, the output of the NOR gate is guaranteed to be low during the φ_1 period. The NOR gate output is then run through an inverting super-buffer, so that during φ_1 the output is guaranteed to be high. At the end of φ_1, the OP code are driven onto the P, K, and R control lines. The only interface specification for the ALU which 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 φ_1, and that the A and B inputs likewise be valid by the end of φ_1 and be stable throughout φ_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 φ_2.

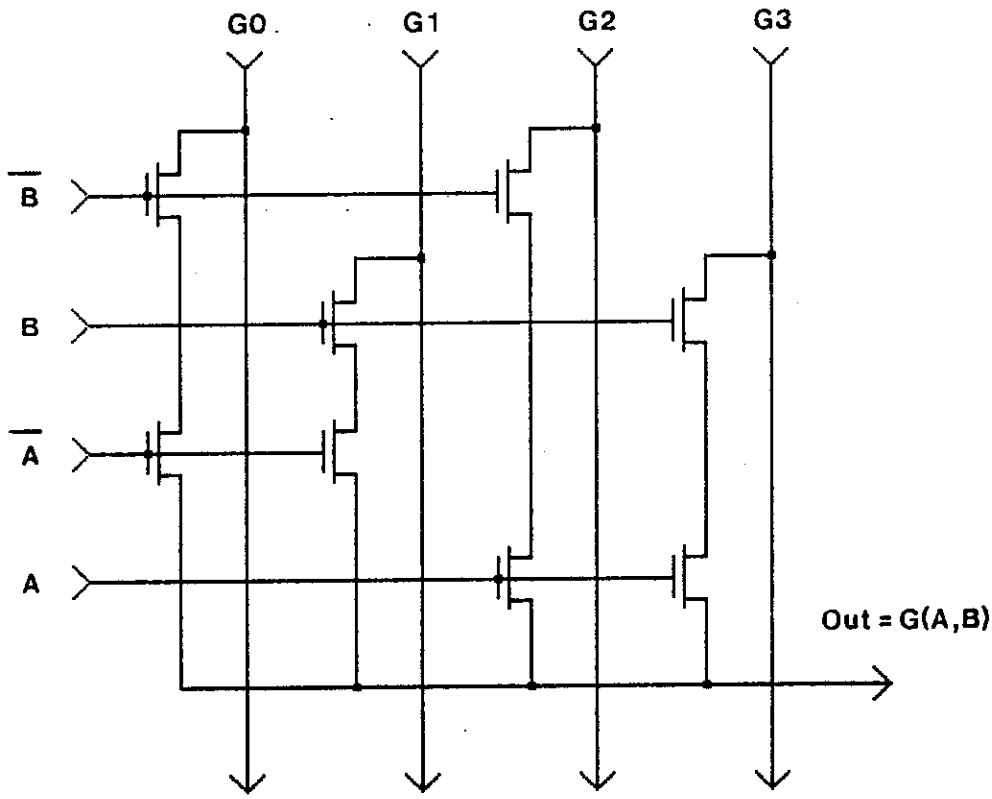


Figure 4. General Logic Function Block Transistor Diagram.

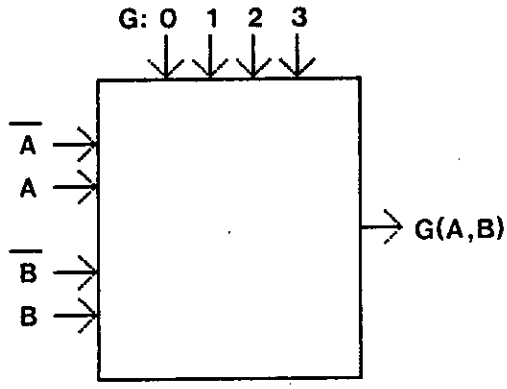


Figure 5. Functional Abstraction of the General Logic Function Block.

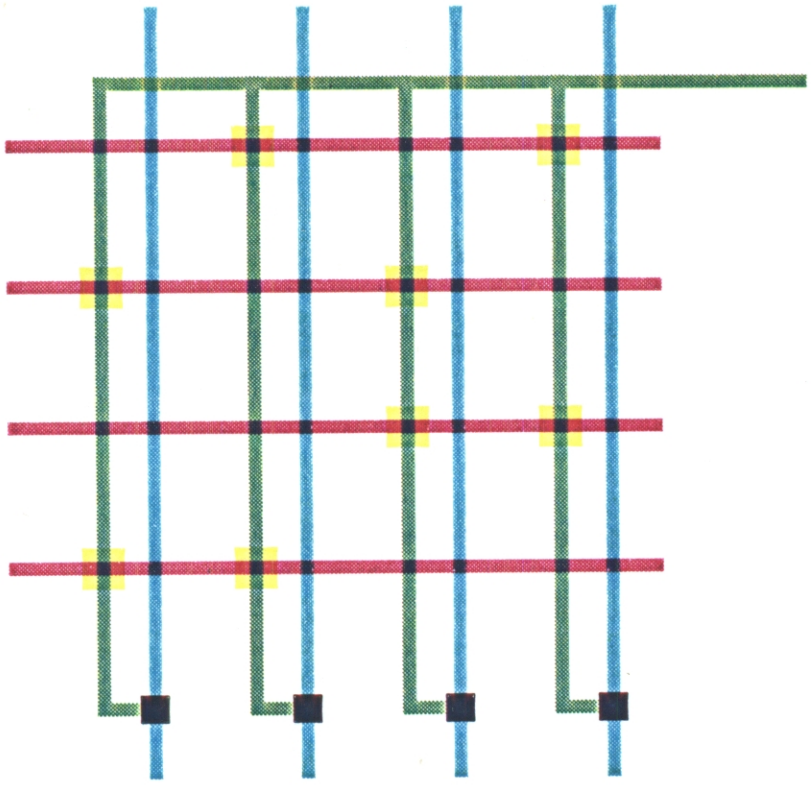


Fig 4a. Stick Diagram of the Function Block

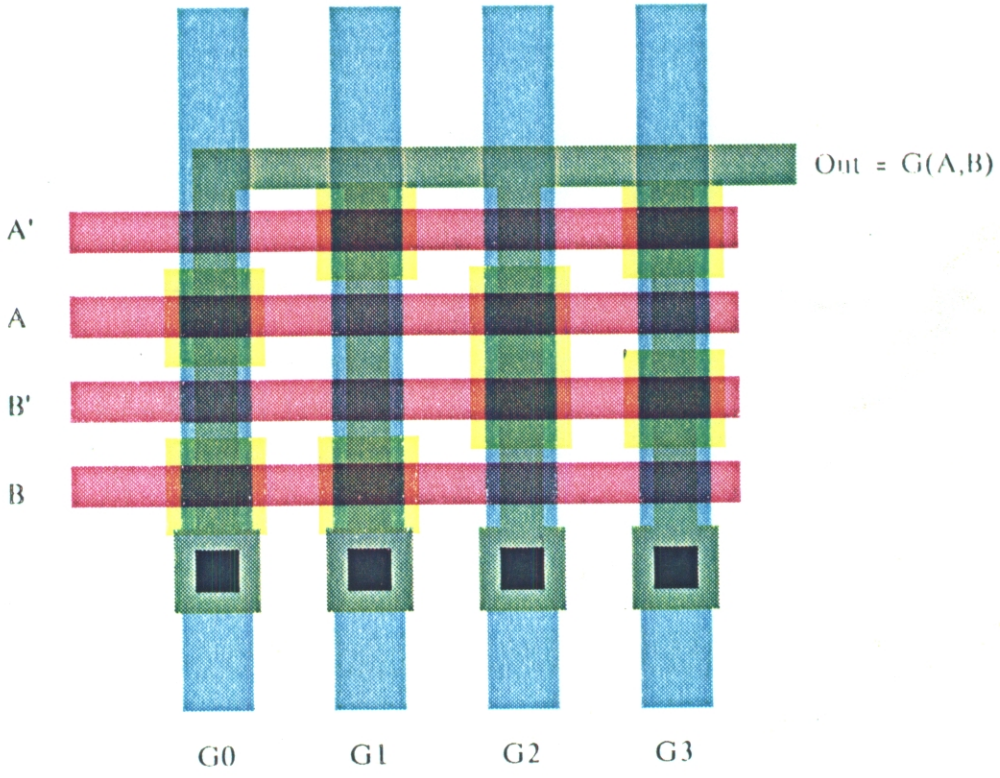


Fig 4b. Actual Layout of the Function Block

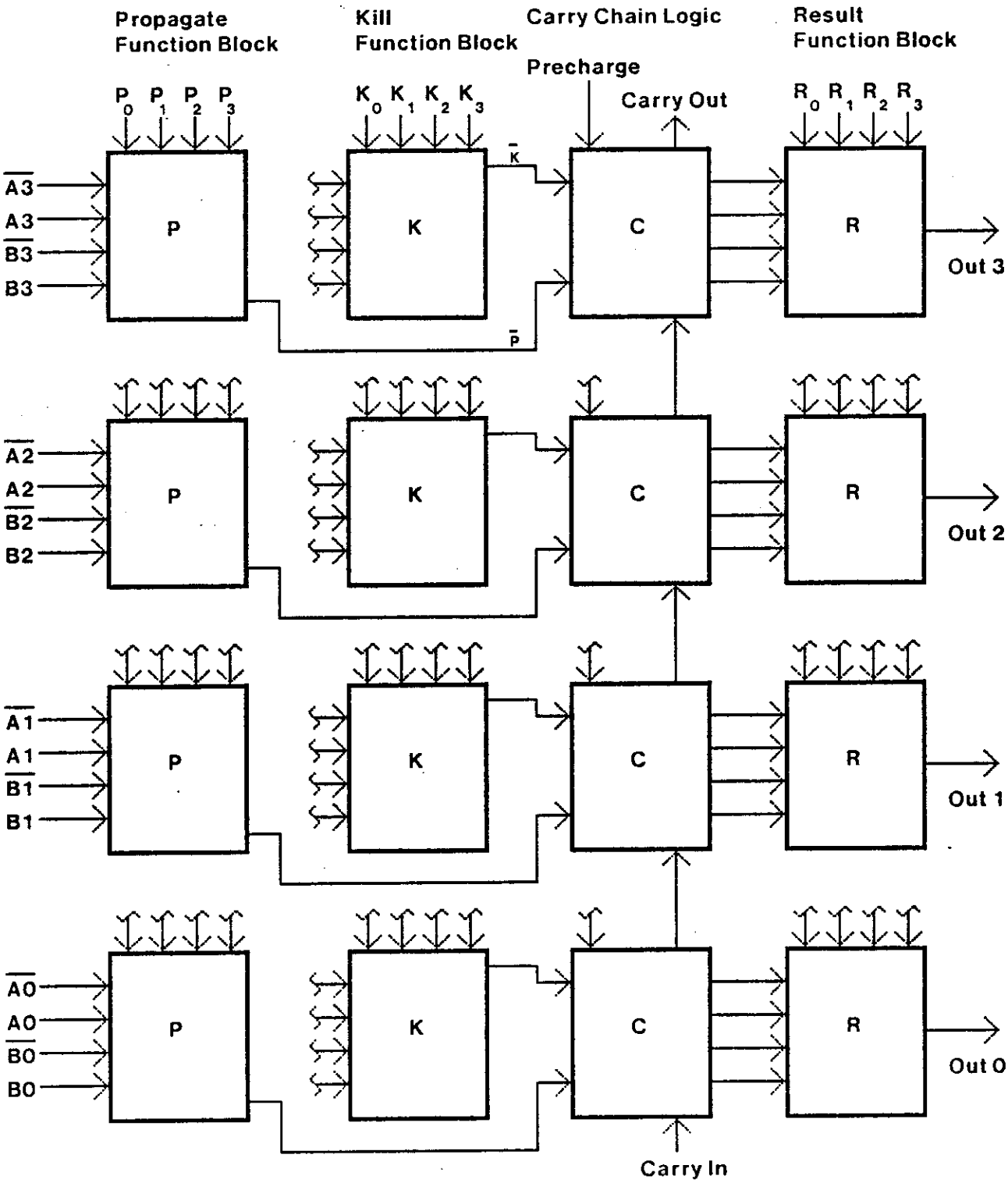


Figure 6. Block Diagram of a 4-Bit ALU.

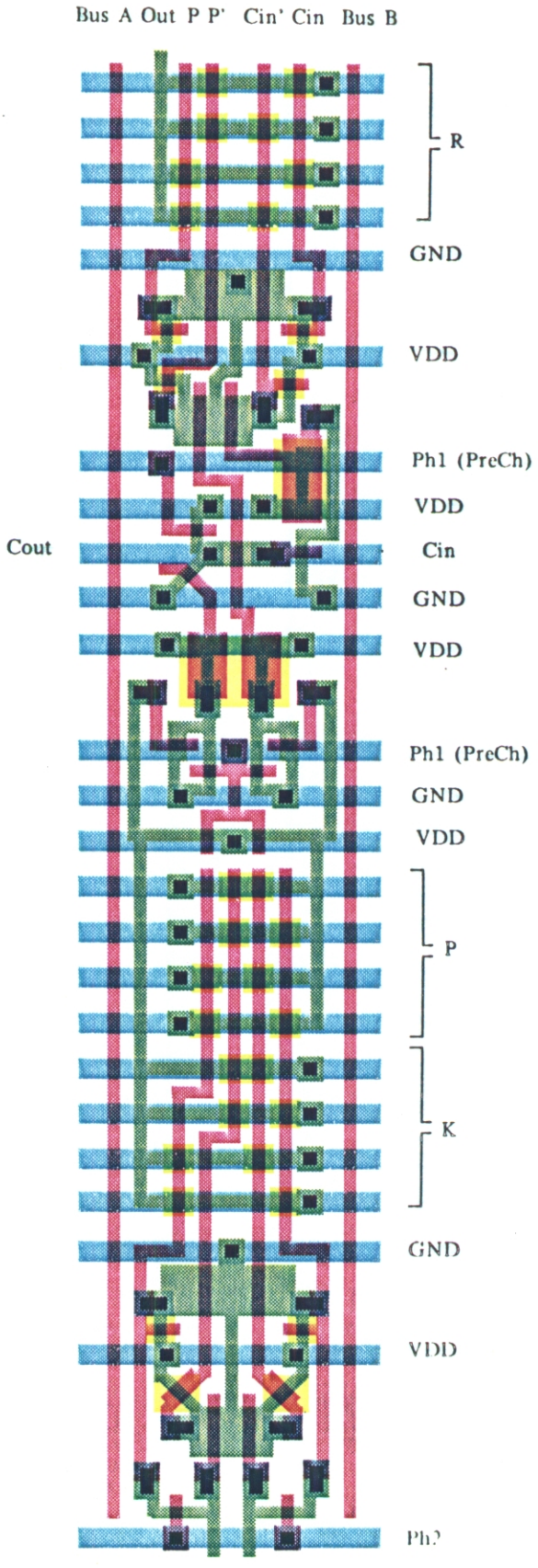


Figure 6a. Layout of ALU Bit Slice and Input Registers

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 $\varphi_1 \cdot \varphi_2$ 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 figure 8.

The input to the first inverter can be derived from four sources: three internal 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 high during φ_1. The feedback transistor around the two inverters is always activated during φ_2. Thus, with three vertical control wires plus the φ_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 φ_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 which selects which source will be applied to the ALU input register during φ_1 must come in during the previous φ_2. Each of the select signals must be low during φ_2, and at most one of them may come high during the following φ_1. A driver appropriate for these control signals is shown in figure 9. The control OP code is latched during φ_2, during which time the NOR gate shown disables the output driver. Since the output driver in this case is non-inverting, the output select line is held low during all of φ_2. At the end of φ_2, the OP code signal is latched and 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 φ_1 and the ALU input selection code in during φ_2, as shown in figure 10. This technique was suggested by Ivan Sutherland.

The ALU output register is similar to the ALU input register, except the timing is reversed. The result of the ALU operation is available at the end of φ_2.

An OP code bit will, if desired, enable the latch signal to go high during φ_2. The feedback transistor is always enabled during φ_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 φ_2 period. Once again, both the output and its complement are available if desired.

Buses

An early design decision was made to have data flow through the data path chip on *two buses* which communicate with all of the major blocks of the system. We have already seen that the ALU performs its operation during the φ_2 period and does not have valid data to place into its output register until the end of φ_2. If data are to be transferred from the output register of the ALU to its input register, this must be done during the φ_1 period. If we adopt a standard timing scheme in which all transfers on the buses occur during φ_1, we can make use of the φ_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 φ_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 tristate 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 tristate drivers become simply two series transistors as shown in figure 11.

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 φ_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 φ_1 period, the bus can be driven from as many sources as necessary. The performance of such a bus is limited only by the pull-down capability of the two series transistors. We shall adopt this philosophy for the processor chip we are designing, and attach such a tristate driver to each of the output registers for the ALU.

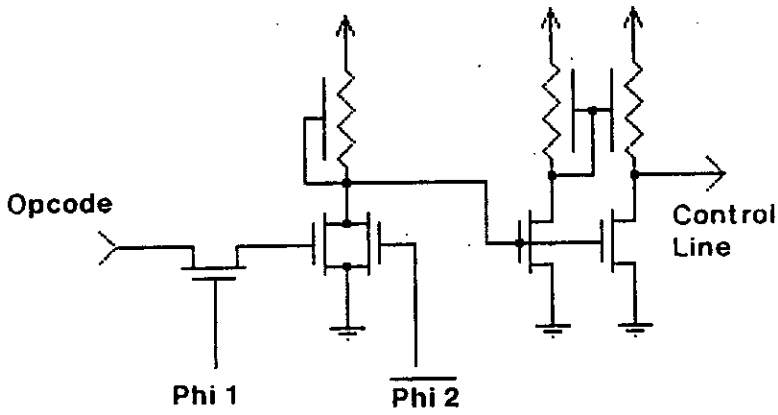


Figure 7. ALU Control Driver

All outputs high during Phi 1
(Precharge)
Selected terms low during Phi 2
Opcode valid during Phi 1

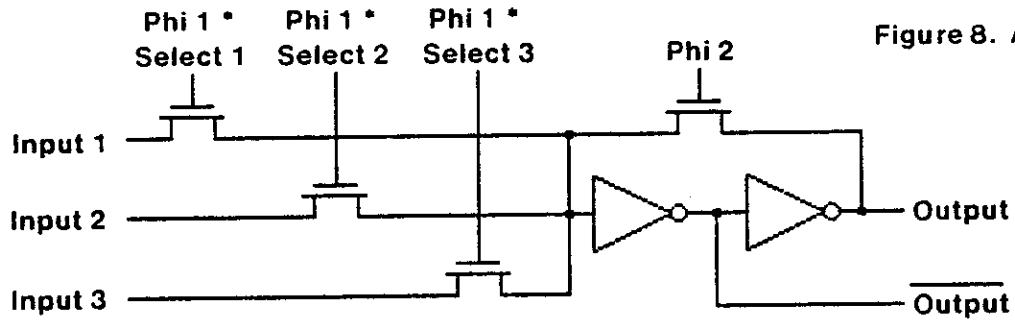


Figure 8. ALU Input Register
and Multiplexer.

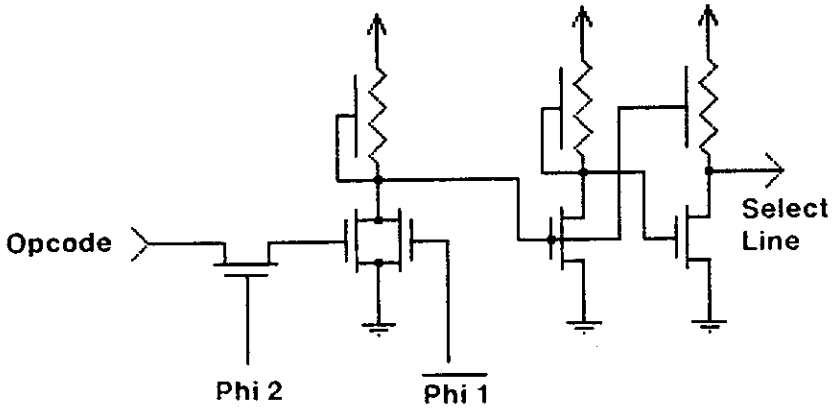


Figure 9. Select Control Driver.

All outputs low during Phi 2
(Precharge)
Selected terms high during Phi 1
Opcode valid during Phi 2

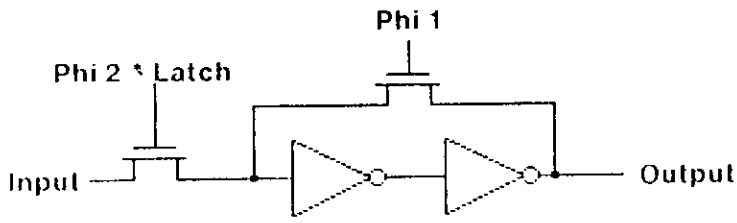


Figure 10. Output Register

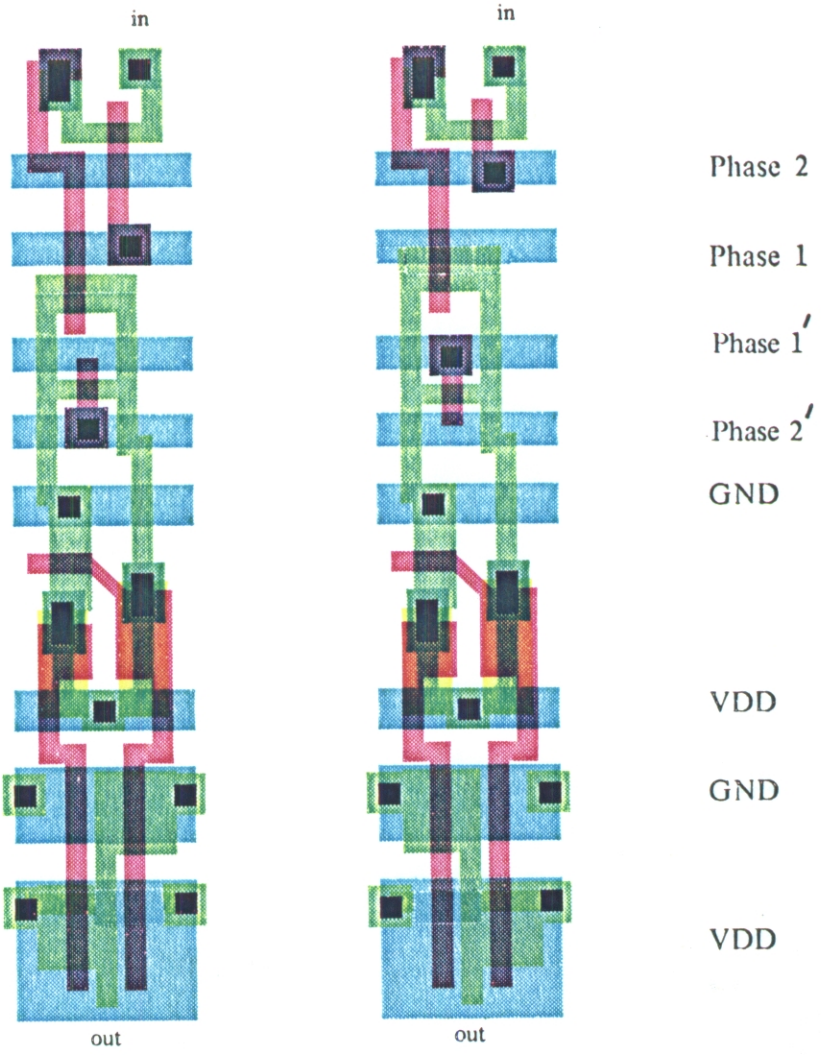


Fig. 7a. ALU Control Driver Layout

Fig. 9a. Select Control Driver Layout

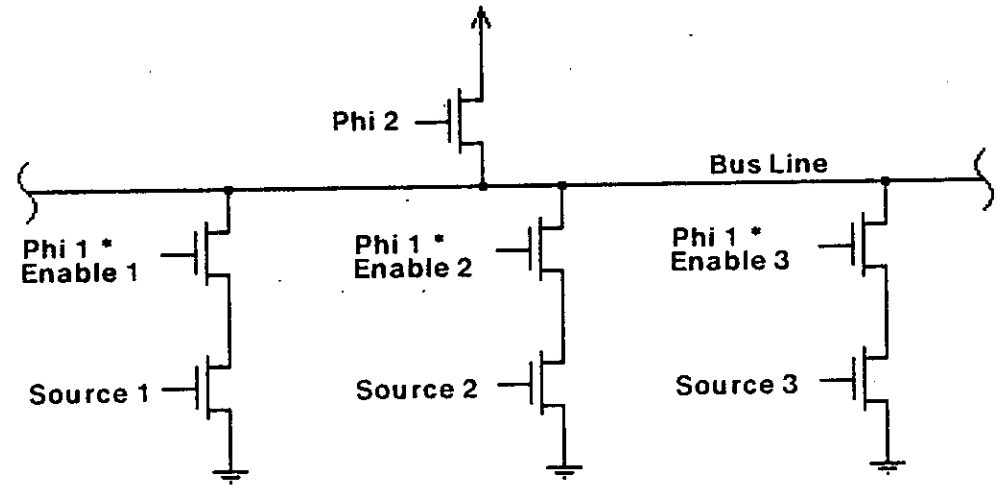


Figure 11. Precharged Bus Circuit.

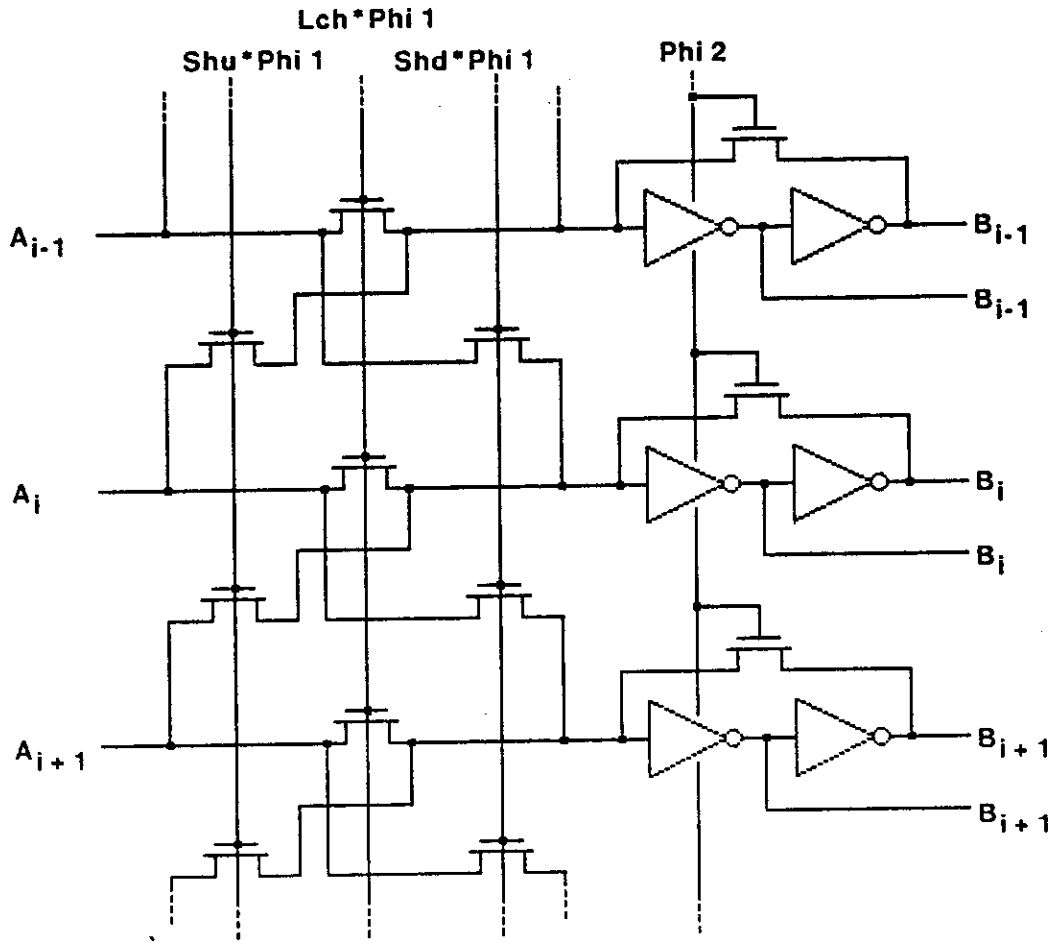


Figure 12. A Simple 1-Bit, Right-Left Shifter.

Barrel Shifter

Since shifting is basically a simple multiplexing function, it might be thought 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 figure 12.

It is identical with the three-input ALU register, and the three inputs have been used to select between the bus, the bus shifted left by one, and the bus shifted right by one. To support the 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 mentioned in Chapter 1 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 figure 13a. 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 shift them, and is completely general in the way in which it can scramble output bits from any input position. In order to maintain this complete generality, the control of the crossbar switch requires n^2 control bits. In some applications, this 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 figure 13b. 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 only required to pull down the shifter's outputs when the appropriate bus is pulled low by its tristate drivers. 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, 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 figure 13b that the bus is available running vertically

through the shift array. It 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 figure 13b 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 extract a word which gracefully crosses the boundary between two data path words. The arrangement shown in figure 13b 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 which are required for the balance of the data path anyway, we may construct a shifter which places the words formed by the two buses end to end and extracts a full-width word which is continuous across the word boundary between the A and B buses. This function is accomplished in as compact a form as just described with a circuit shown in figure 14. Notice that the vertical wires have a split in them. The portion of the wire above the corresponding shift output being connected to the A bus, and that below the corresponding shift output to the B bus.

It can be seen by inspection that this circuit performs the function shown in figure 15 which is just what is required for doing field extractions and variable word length manipulations. The literal port is connected directly to the A bus and may be run backwards in order to discharge the bus when a literal is brought in from the control port. A block diagram which represents the shifter at the next level of abstraction is shown in figure 16.

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

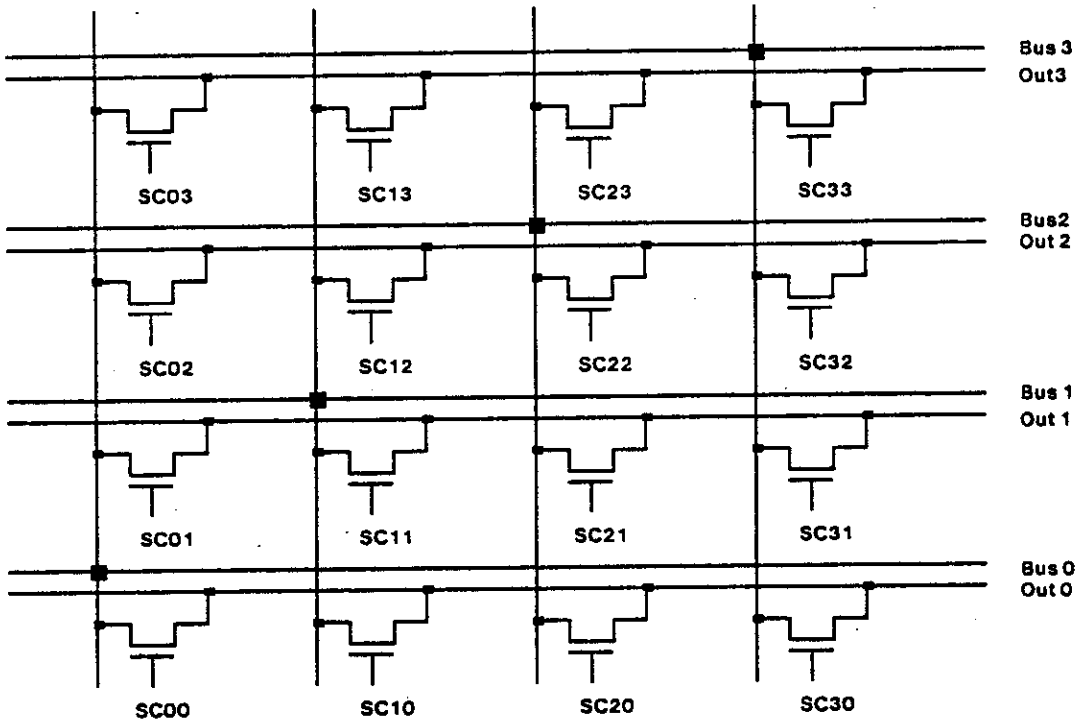


Figure 13a. 4-by-4 Crosssbar Switch

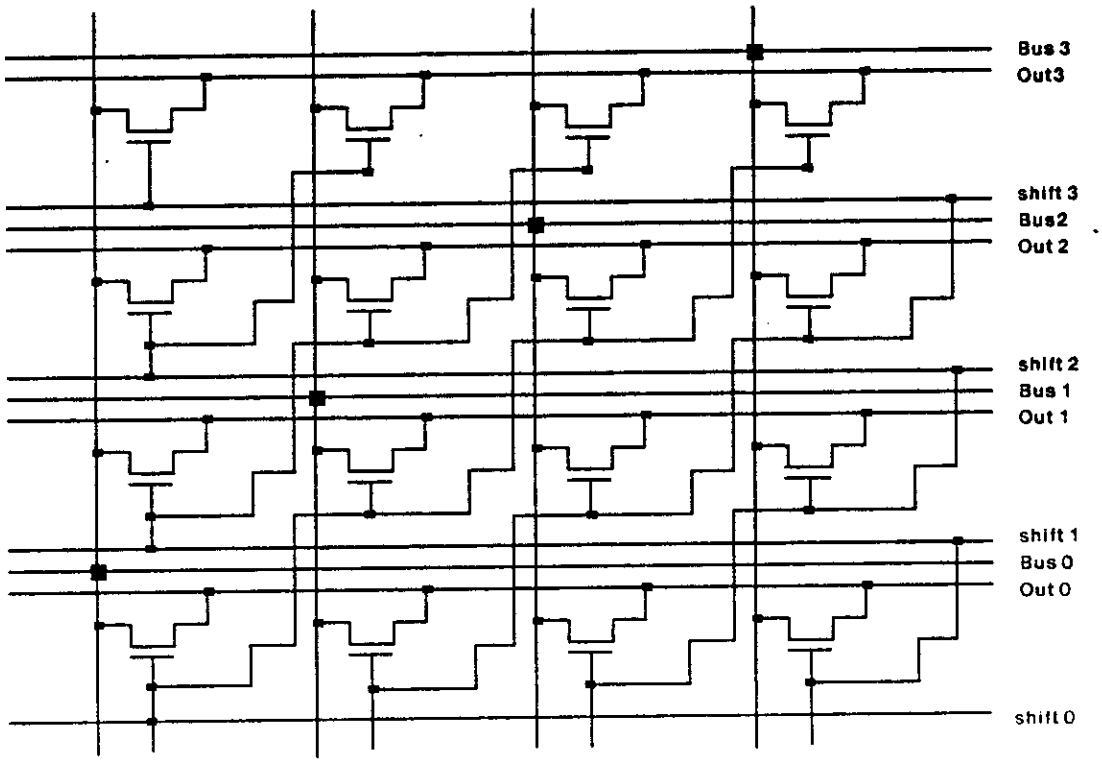


Figure 13b. 4-by-4 Barrel Shifter

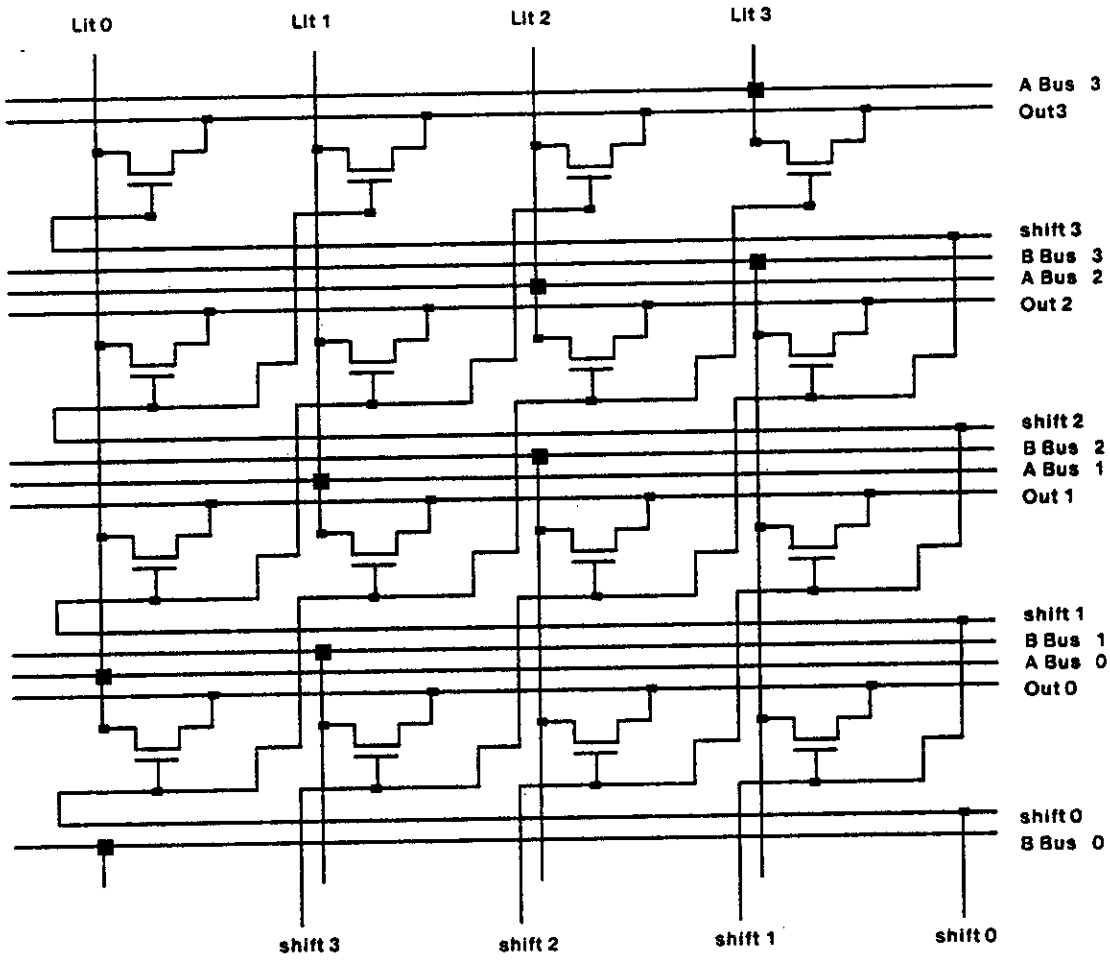


Fig.14. 4-by-4 Shifter with Split Vertical Wires and 2 Data Buses

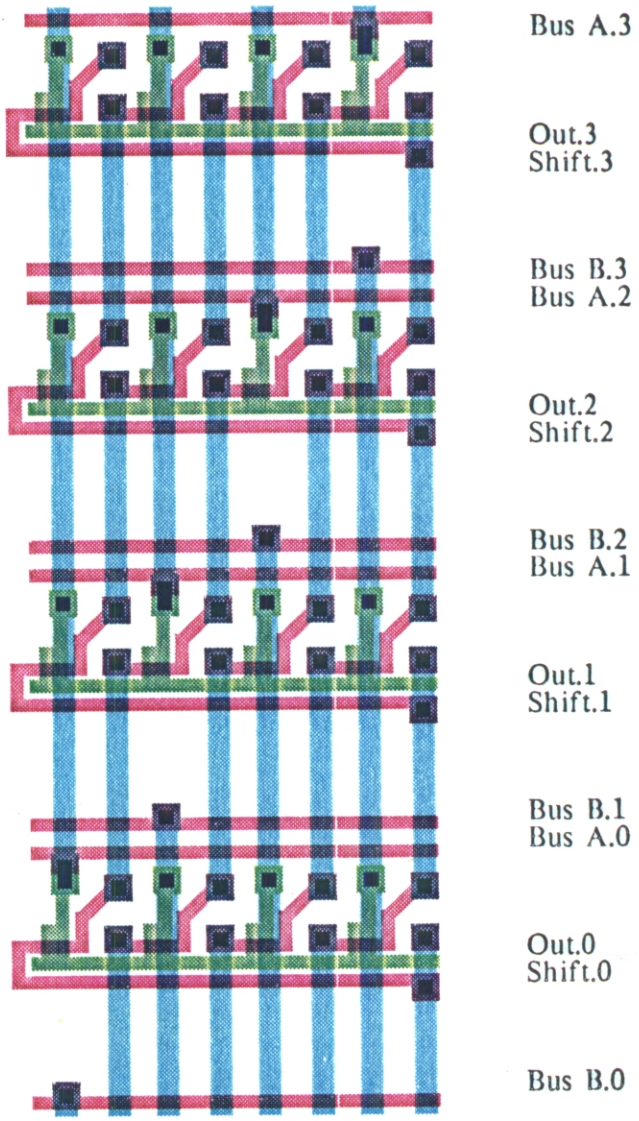


Fig. 14a. Layout of a 4-Bit Barrel Shifter

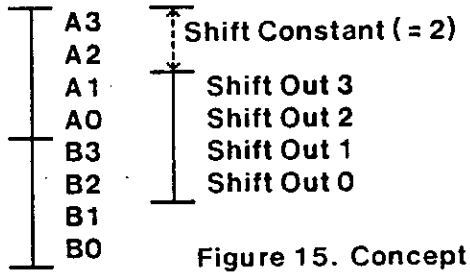


Figure 15. Conceptual Picture of the Shifter's Operation.

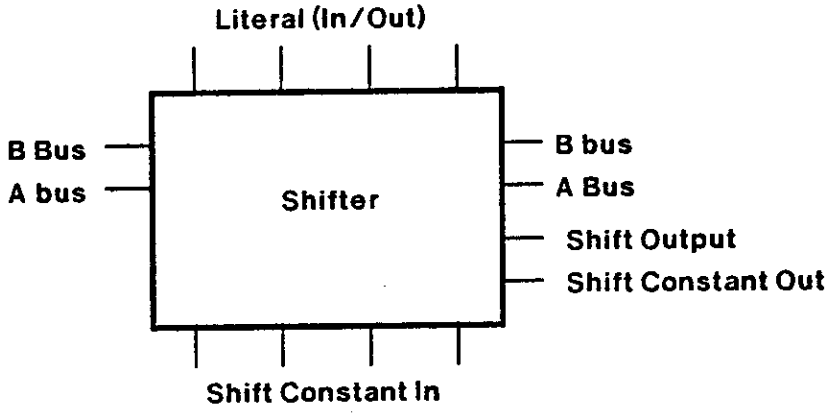


Figure 16. Block Diagram of the Shifter.

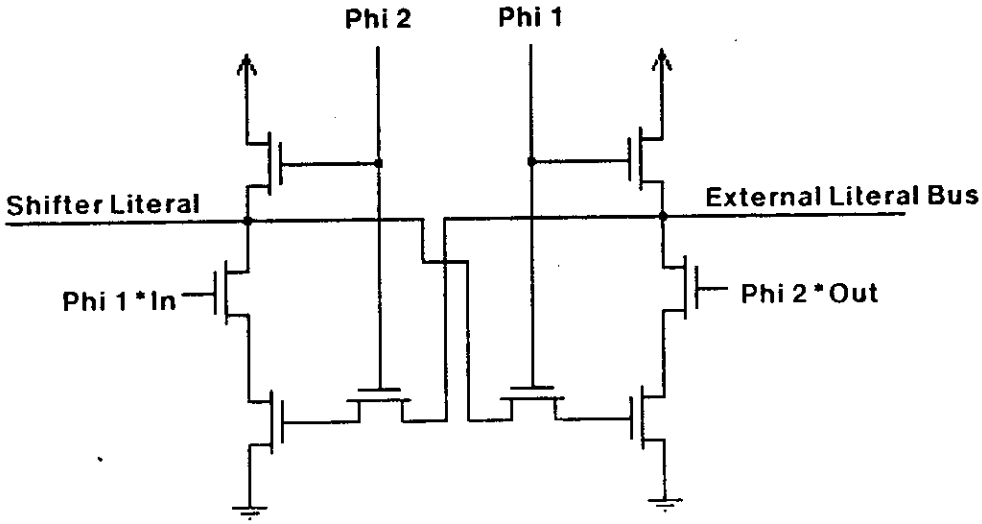


Figure 17. Literal Interface.

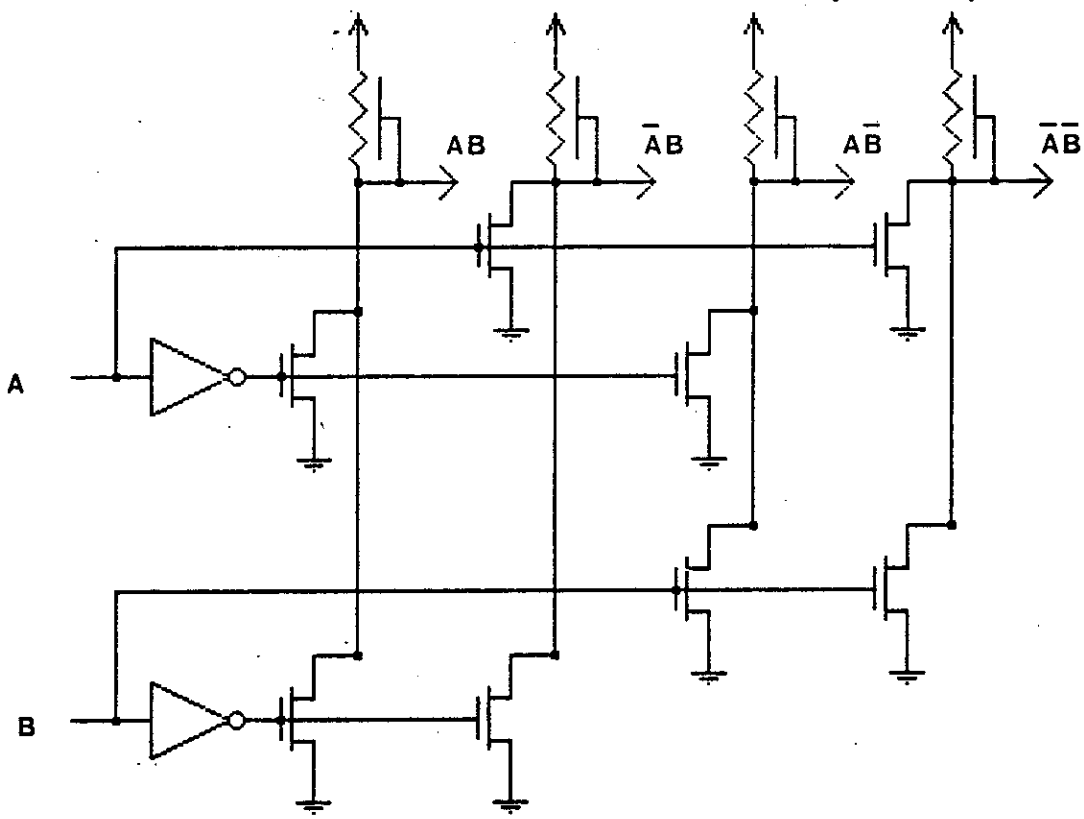


Figure 18. A Nor Form 1-of-N Decoder.

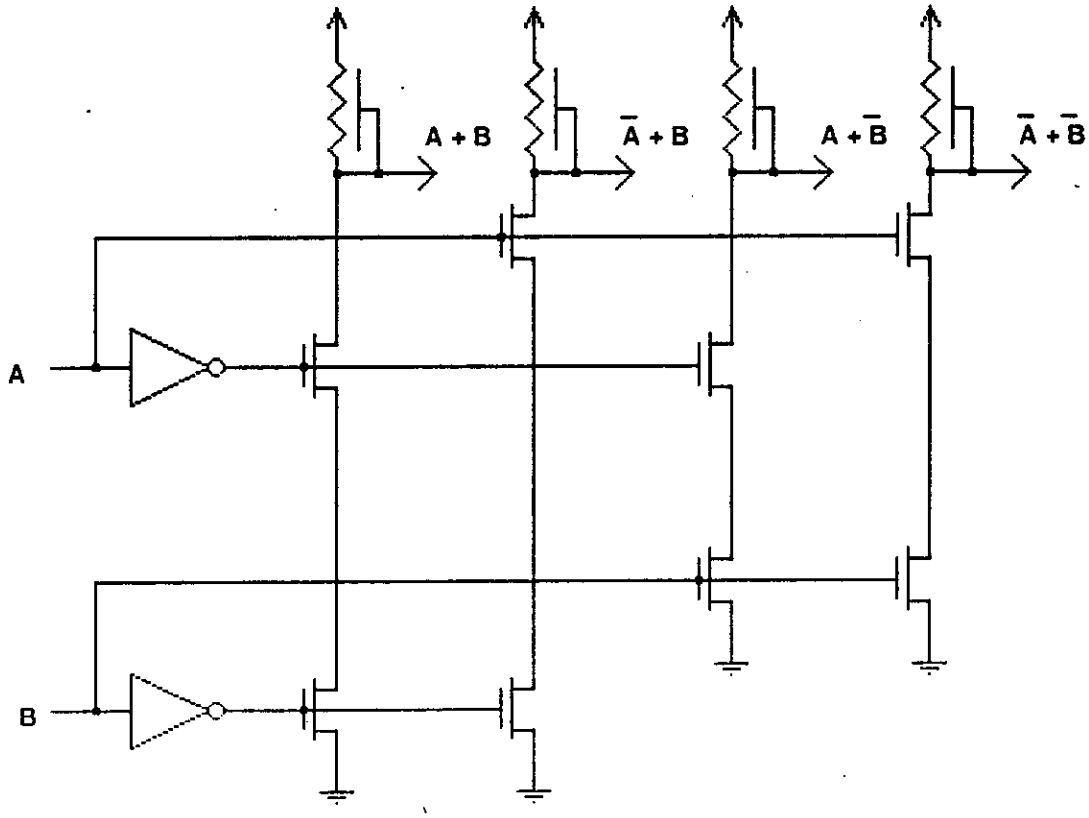


Figure 19. A Nand Form 1-of-N Decoder.

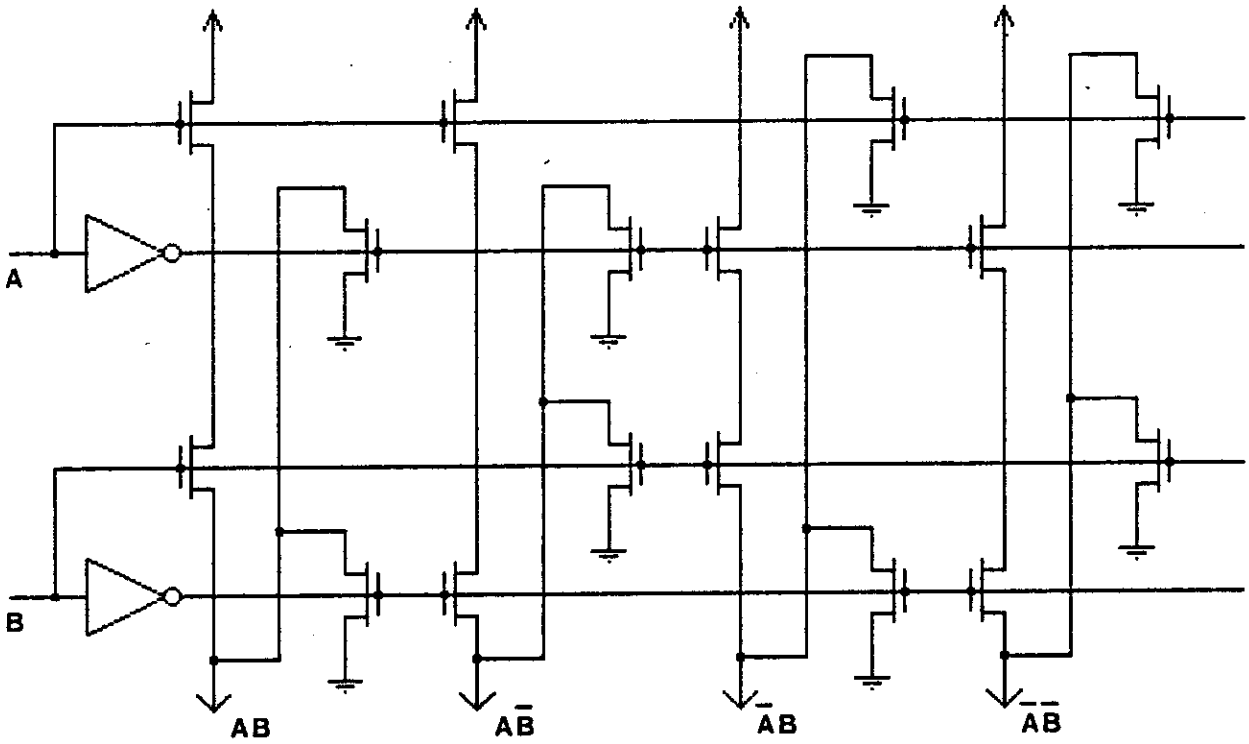


Figure 20. A Complementary Form 1-of-N Decoder.

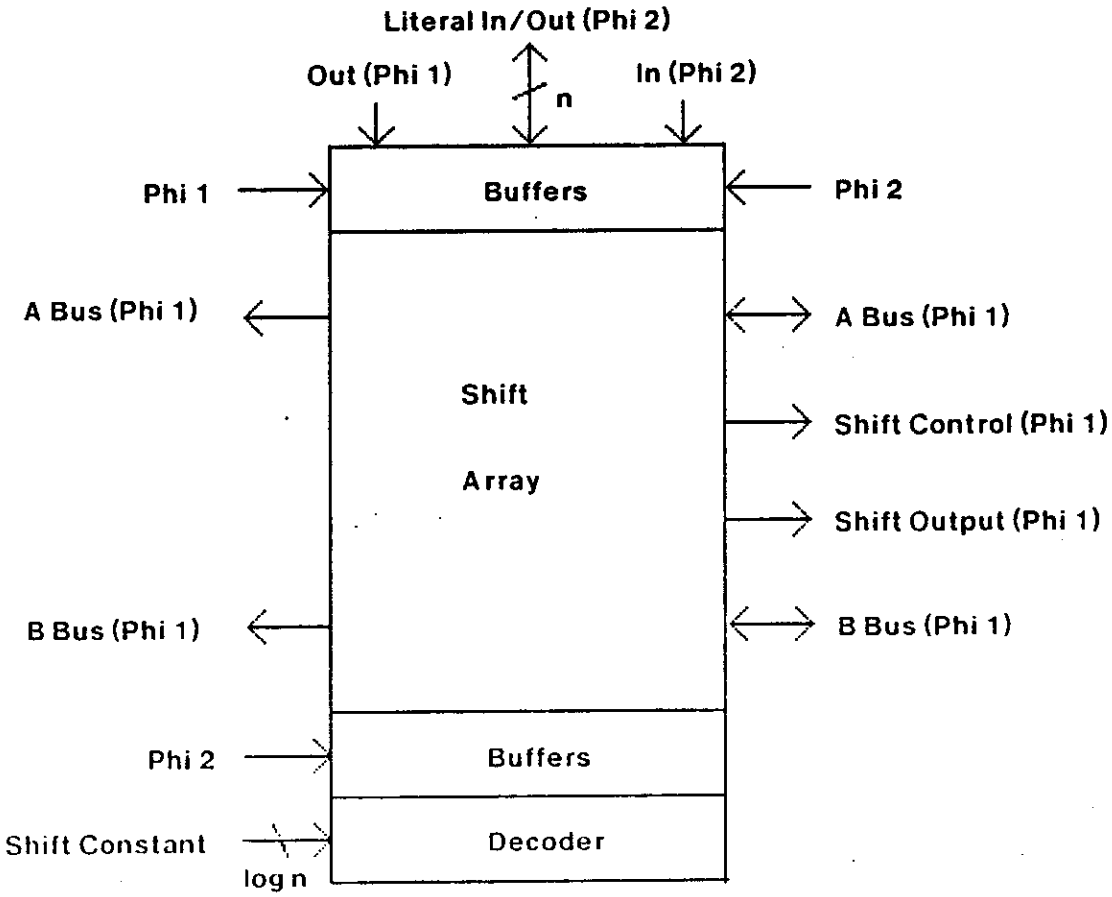


Figure 21. A Fully Synchronized Shifter.

which will operate in either direction is shown in figure 17.

The internal shifter output is precharged during φ_2, and active during φ_1. It may be sourced either from the literal bus or from the shifted combination of the A and B buses through the shift array, shown in figure 15. The external literal bus itself may be sourced either from the opposite end (the external paths from the program source) or from the end attached to the A-Bus in the shift array shown.

The bus to the external literal path is precharged during φ_1, and data bits from the literal port of the shifter are enabled onto it by a signal active during φ_2, as shown in Fig. 17. The two signals, φ_1 * IN, and φ_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 figure 9.

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 to convert the binary number into a one-of-n signal suitable for feeding the buffers. Decoders can be made in a number of ways in the ratio technology we are discussing. 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 figure 18. Notice that the output is a high-going one-of-n pattern.

Decoders can also be made in other forms. For small values of n, the NAND form shown in figure 19 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 lowgoing one-of-n pattern. There is also a complementary form of decoder which can be built with ratio technology, and was 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 fully complementary technologies, and has similar benefits. It can generate either a highgoing or a lowgoing one-of-n number, and dissipates no static power. A decoder of this sort is shown in figure 20. Once we have added the appropriate buffers and decoders to our shift array, we have a fully synchronized function block ready to be integrated with the system at the next level up. The properties of this block are shown in figure 21.

Register Array

In any microprogrammed processor designed for emulating an instruction set at a higher level, it is convenient to have a number of miscellaneous registers available, both for working storage during computations and for storing pointers of specific significance in the machine being emulated: stack pointers, base registers, program counters, etc. Since the data path has two buses, and the ALU is a two-operand subsystem, it is convenient if the registers in data path are two-port registers. Using the design philosophy we have been discussing, a typical two-port register cell is shown in figure 22. This register is a simple combination of the input multiplexer described earlier, the φ_2 fedback transistor, and two tristate output drivers, one for each bus. The registers can be combined into an array m bits long and n bits wide, the buses passing through the array. Each cell of the array is defined at the next level up, as shown in figure 23. Drivers for the load inputs and the read outputs are identical to those shown in figure 9. 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 the A bus besides the registers, and we will conserve microcode bits by encoding them together.

Before we proceed, there is one mundane detail which must be taken care of in the overall topological strategy. The routing of VDD and ground 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 functional blocks. Otherwise, one is apt to be faced with topological impossibilities because certain conductors placed for other reasons interfere with the routing of the VDD and ground. A possible strategy for the overall routing of VDD and ground paths is shown in figure 24.

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 strategy will do, but it must be consistent, thoroughly thought through at the beginning, and rigidly adhered to during the execution of the project.

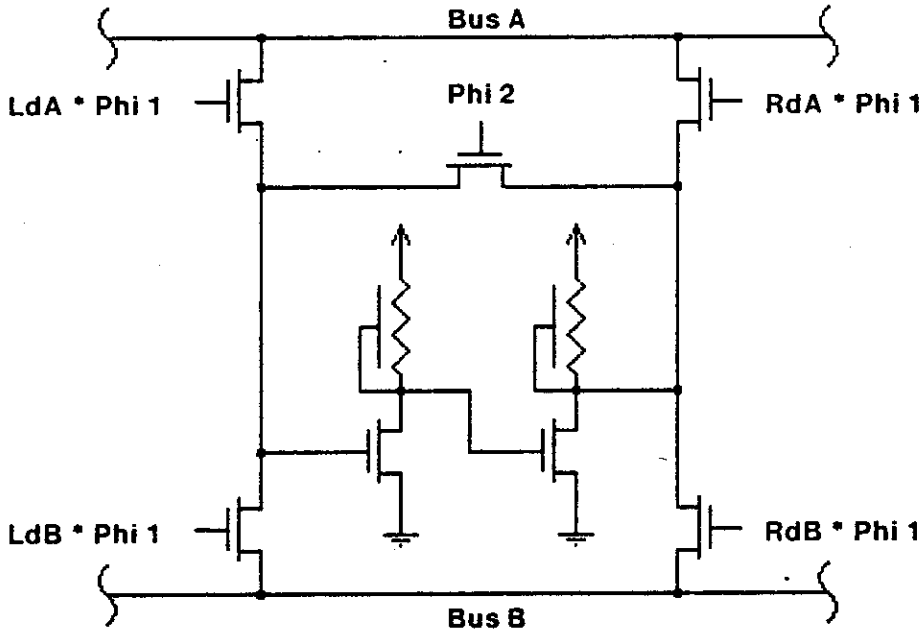


Figure 22. A Two Port Register Cell.

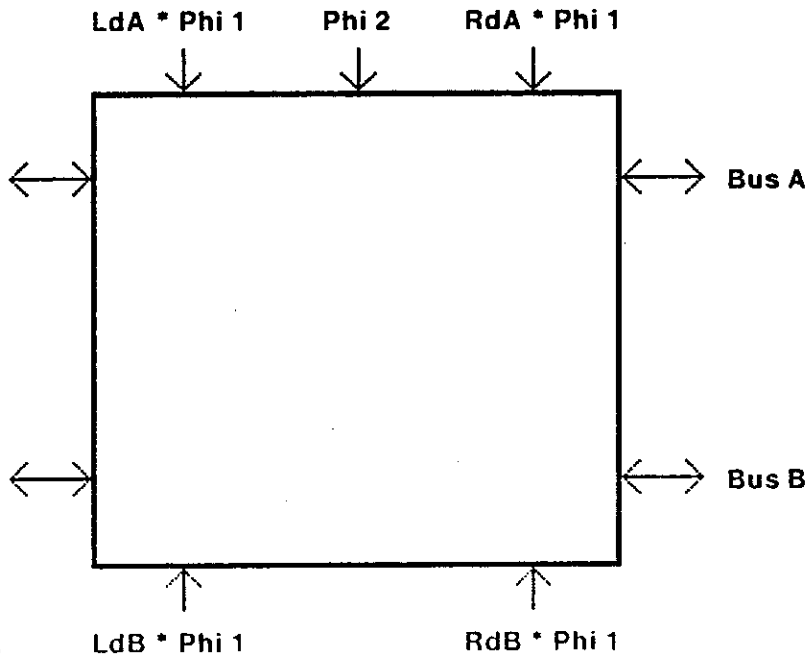


Figure 23. Block Diagram Definition of the Two Port Register Cell.

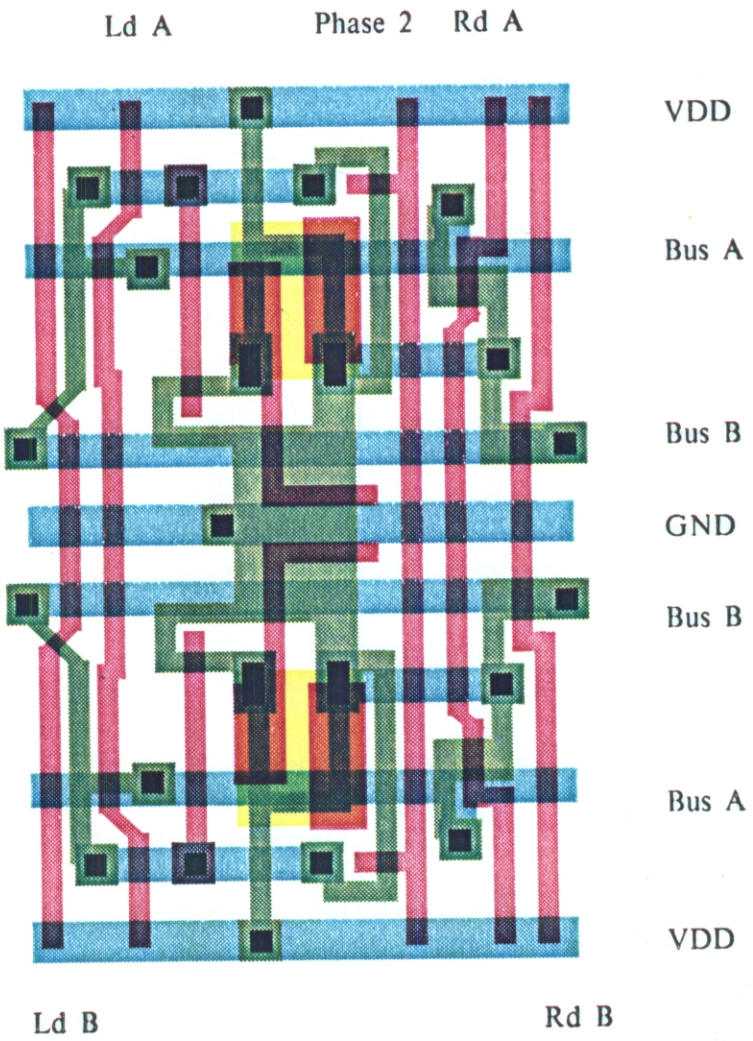


Fig. 22a. Layout of Two Dual-Port Register Cells

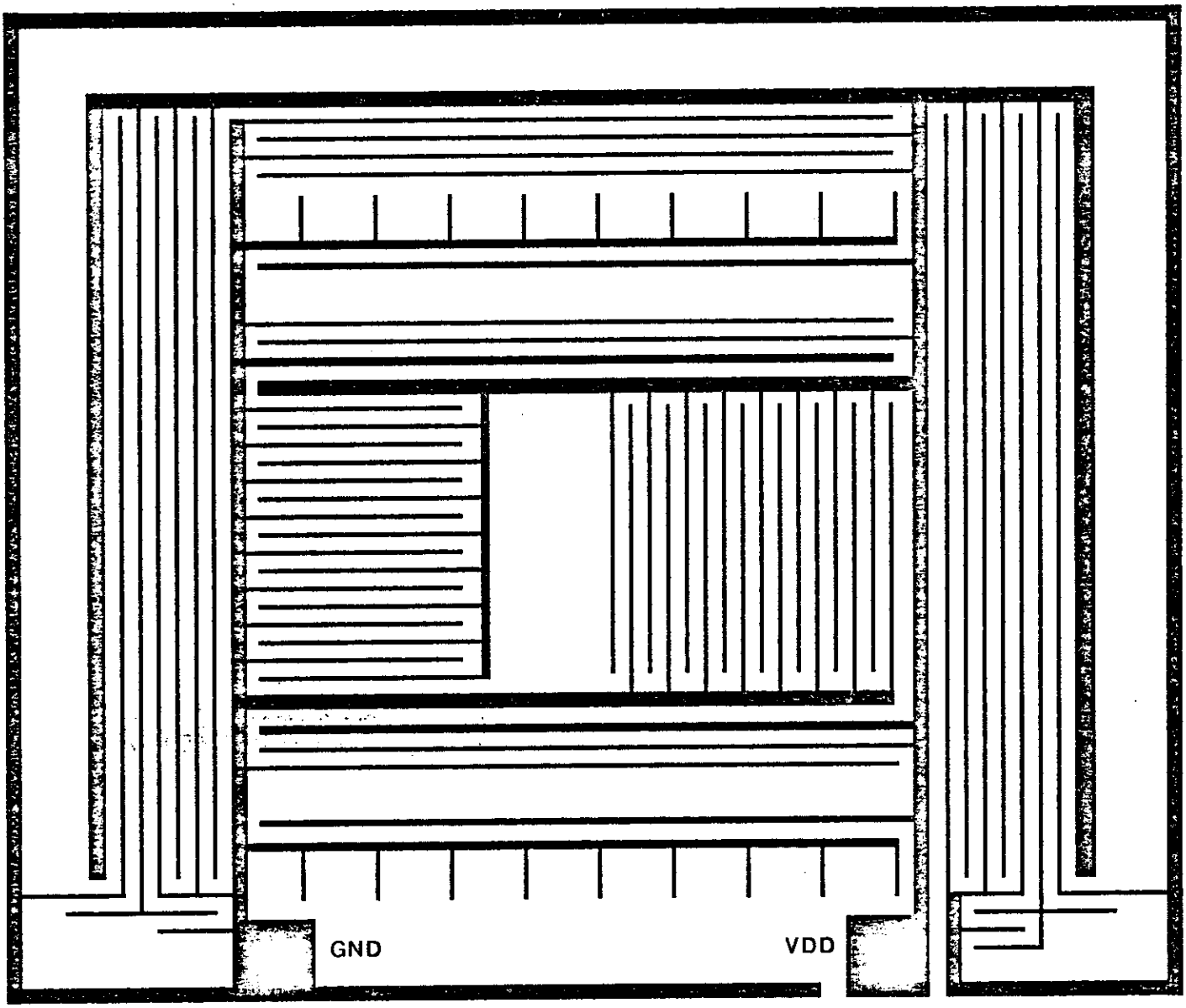


Figure 24. VDD and GND Net for the Data Path Chip.

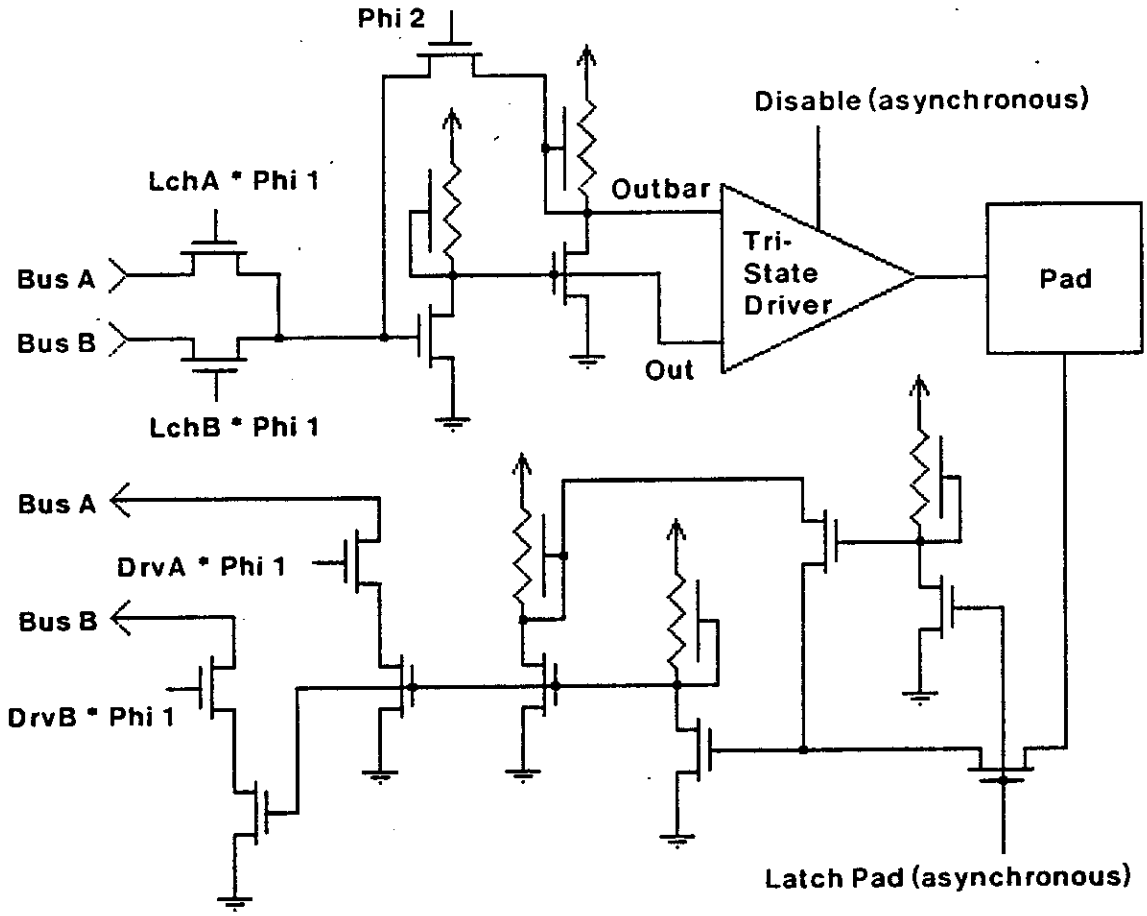


Figure 25. Data Port Tristate Pad Circuit

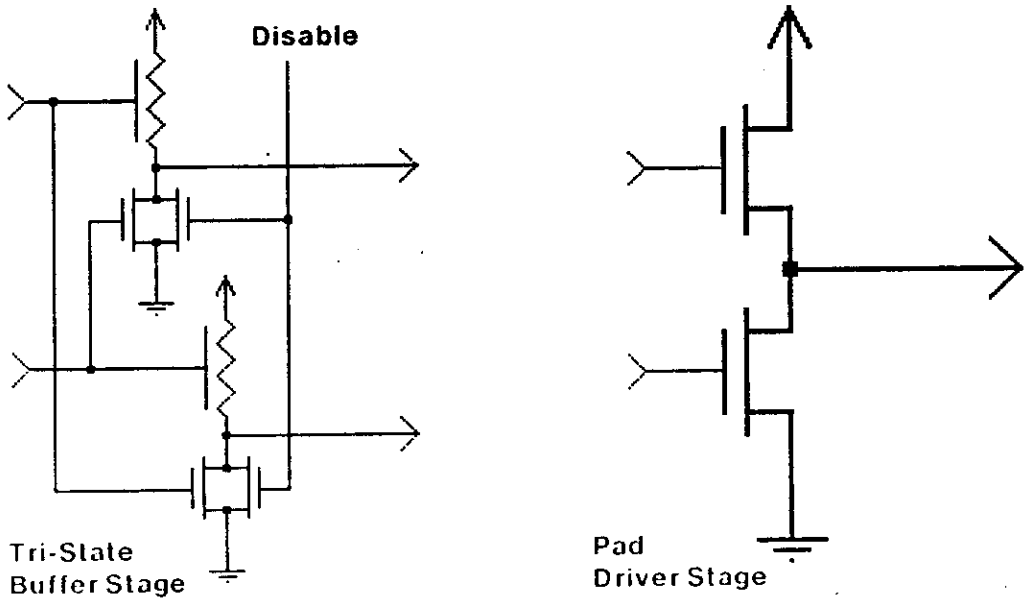


Figure 26. The Tri-State Driver, which consists of any number of Tri-State Buffer Stages followed by a Pad Driver Stage. The Current Design used Two Tri-State Buffer Stages.

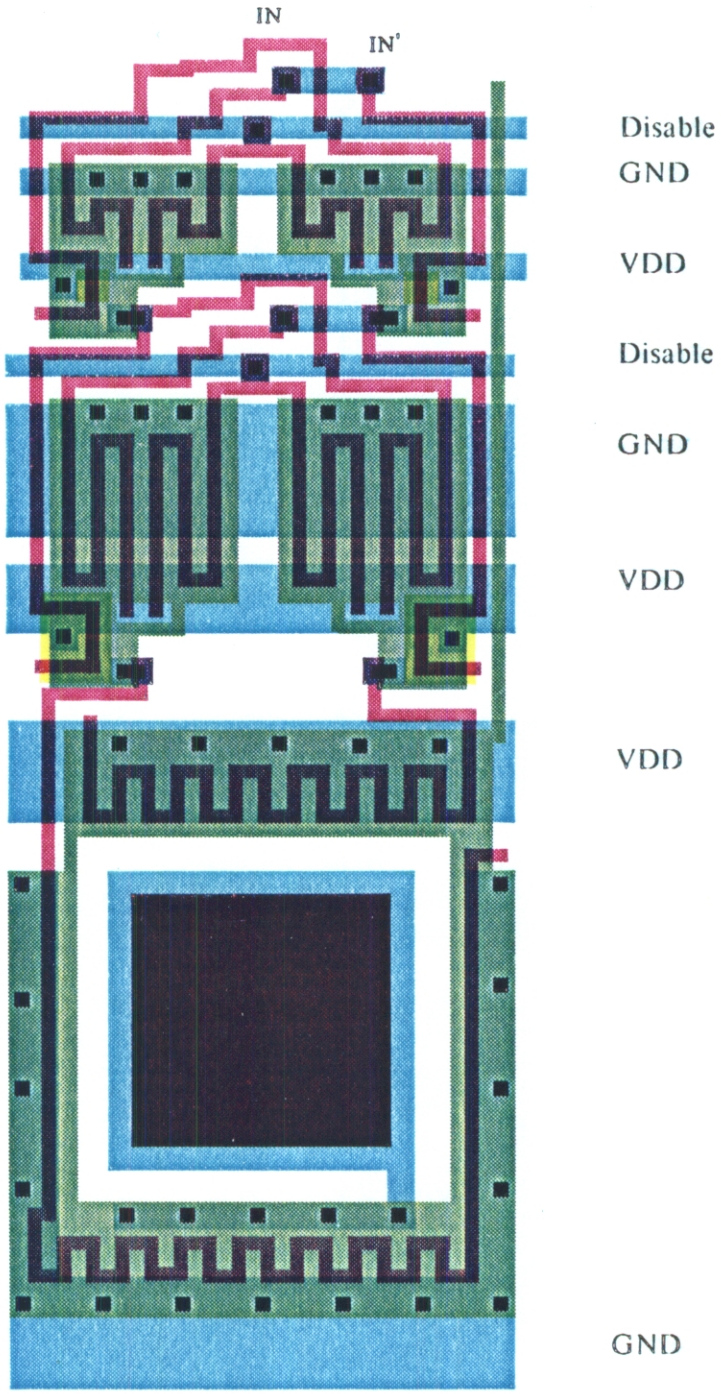


Fig. 25a. Pad Driver Layout

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 to a bus. Thus it is desirable to use port drivers which may be set in a high impedance state. Drivers which can either drive the output high, drive the output low, or appear as a high impedance to the output are known as *tristate* drivers. Such drivers allow as many potential senders on the bus as necessary. Figure 25 shows the circuit for a tristate interface to a contact pad.

Here, either bus A or bus B can be latched into the input of a tristate driver during φ_1. Likewise the pad may be latched into an incoming register at any time independent of the clocking of the chip. Standard tristate drivers are enabled on bus A and B. The only remaining chore is the design of the tristated buffer which drives the pad directly. Details of the tristate driver are shown in figure 26.

The terms out and outbar are fed to a series of buffer stages which 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, but reduces the current to a value where it can be handled by the disable transistor of the following buffer stage. 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. 26b. 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 tristated. Once again the two output transistors are a factor of approximately e larger than the last super buffer in the buffer string.

As we have seen, the inverter string necessary 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 is quite large, and imposes a delay of some factor times a logarithm of this impedance ratio upon communications between the chip and the outside world. Any help which 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 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 thus the total delay in going off chip.

Data Path Control Operation Encoding

By now we have defined a complete functional data path with ports on each end and functional blocks through the center, as shown in figure 27. The data path operation code bits required to control the data path and the phase of the clock on which they are latched are shown. There are forty-nine such bits together with 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 port 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 micro program counter and controller, in which case all operations which can be done by the data path may be done in parallel. The opposite extreme is to very 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 which would then decode the compact microinstruction into the data path operation code bits.

The important point of the design strategy we have chosen 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 very well defined, very clean, and 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. As time progresses and it is possible to construct chips with larger and larger functional density, blocks of the sort shown will form components in even larger geometrical arrangements which will form even larger components and a whole hierarchy will emerge which will implement a system function at a much higher level than contemplated here. However, if the design strategy we have described is followed, it is possible to construct arbitrarily large and complex systems which are guaranteed to work if the individual component blocks are correct, and given the clocking period is sufficient to allow the slowest functional unit to perform its function.

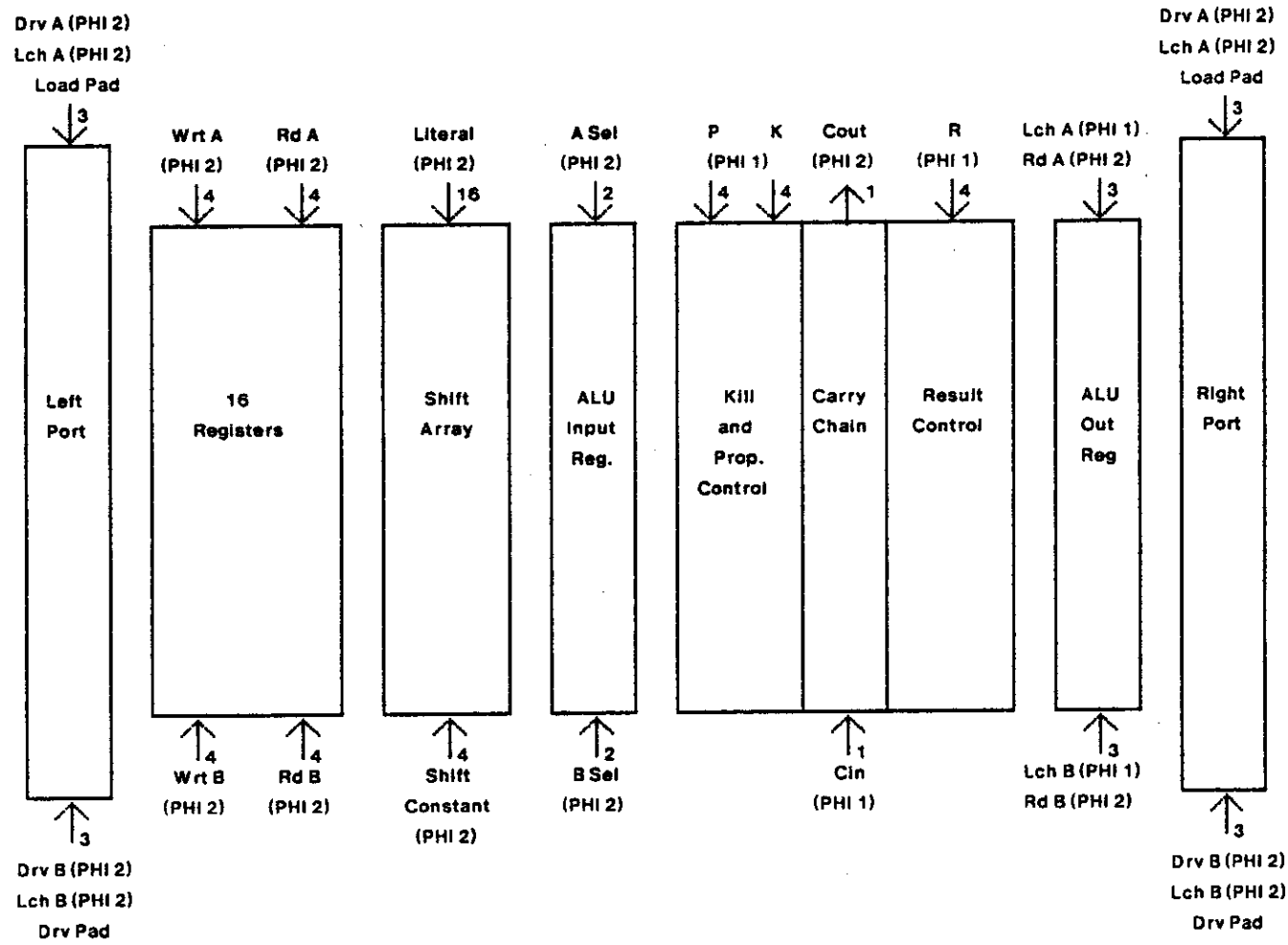


Figure 27. Block Diagram of Datapath with Control Wires Added.

Using the approximate capacitance values given at the end of Chapter 2, an estimate can be made of the minimum clock period for sequencing the data path. The Phase 1 time of the data path is ~50τ, the same as the general estimate given in the section "Transit Times and Clock Periods" in chapter 1. However, the Phase 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 Figure 6a. The metal and diffusion occupy ~15 and ~8 times the area of the propagate pass transistor gate, respectively. Metal is ~0.1 and diffusion is typically 0.2 times the gate capacitance per unit area. Thus the total capacitance of each stage of the carry chain is ~4.5 times that of the pass transistor gate. The effective delay time is correspondingly longer than the transit time τ of the transistor itself. The effective delay through n stages of such pass transistor logic is ~ τn^2. In the OM2, n=4 and the effective delay for 4 bits of carry chain is ~4.5*16τ = 72τ. 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 ~ 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 ~30τ. The total carry time is thus ~100 times the transit time for each block of 4 ALU stages. The total Phase 2 time should then be ~400τ. In 1978, the fastest commercial nMOS processes yield a transit time τ of approximately 0.3 ns, and we would expect a minimum total clock period of ~450τ, or ~135 ns.

The Second Half of this Chapter contains a functional specification of the OM2 data path chip, by Dave Johannsen of Caltech. 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.

Functional Specification of the OM2 Data Path Chip

[Section contributed by David L. Johannsen, Caltech]

Introduction

This specification describes a 16-bit data path chip referred to as OM2 [#986]. The OM2 contains 16 registers, an ALU, and a 32-bit shifter, and is designed as part of a micro-programmed writeable-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 Datachip 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 Datachip the phase before that code is to be executed. Therefore, the bus transfer code enters the Datachip when the clock is low, and the ALU code enters when the clock is high. Figure 1 sketches a possible OM system. For a more detailed description of system configurations, see reference 1.

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.

Datapaths

A block diagram of OM2 is shown in figure 2. There are two buses which connect the various elements of the chip together. These 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 only get data from one source, and give data to one destination during any one cycle.

The Left and Right Ports communicate between the datachip 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 datachip 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 datachip. Inputting from the port is similar. By pulling down on another enable pin, data from the external bus is 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 OM2s. For internal control only, the external enable pins can be left floating.

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 to the use of the registers:

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

Shifter

The shifter concatenates the two buses, resulting in a 32-bit word, with the A bus 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 the B bus present in the output (ie. a shift constant of 0 returns the A bus, while a shift constant of 15 returns the LSB of the A bus in the MSB of the output, followed by all but the LSB of

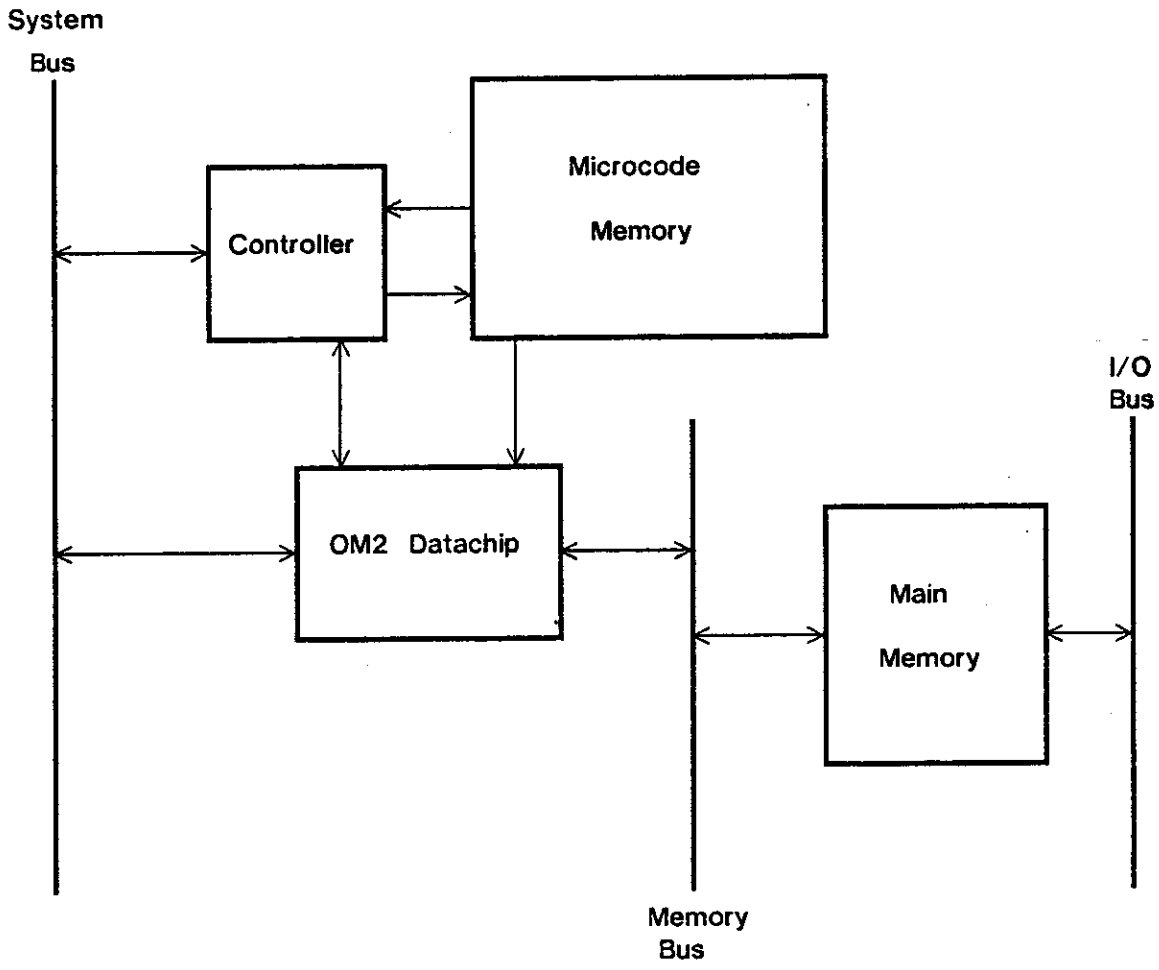


Figure 1. One Possible OM2 System Configuration

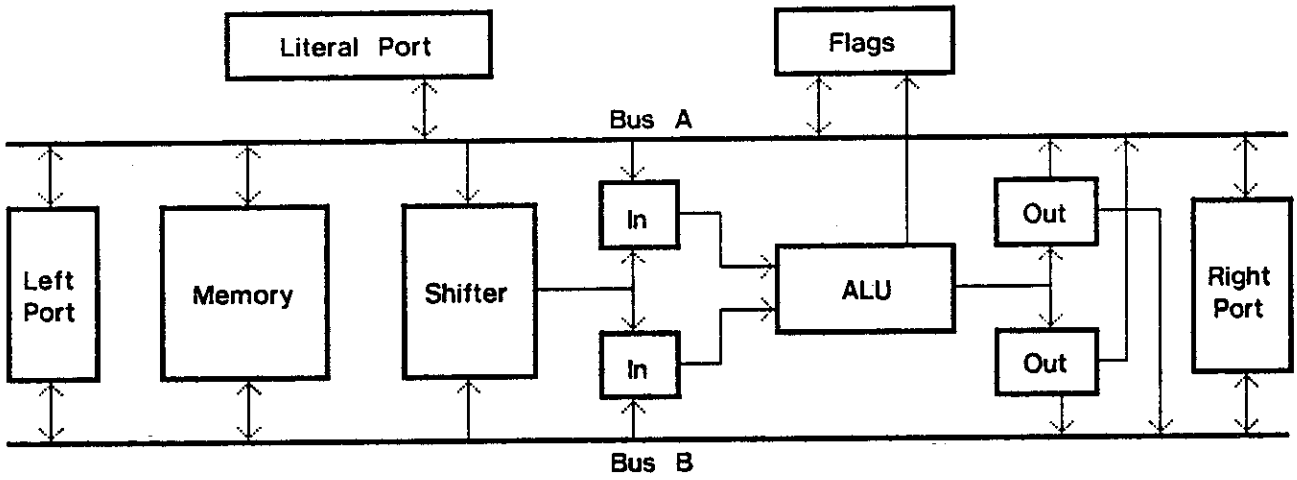


Figure 2. Block Diagram of OM2

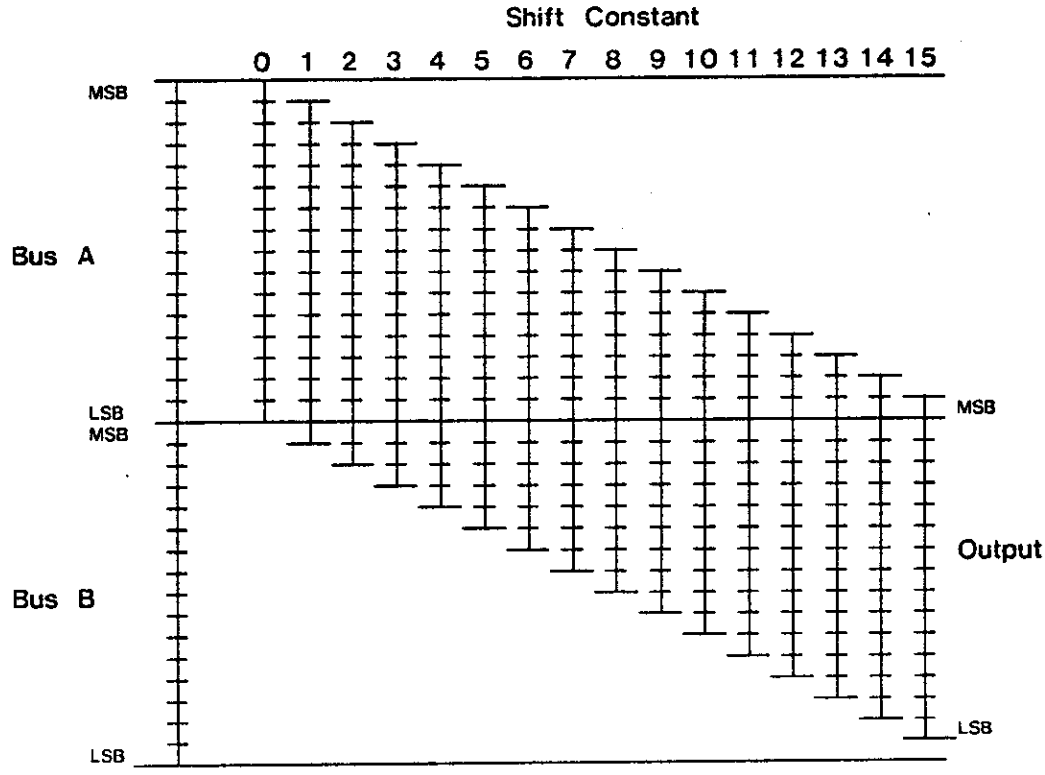


Figure 3. Shifter Operation.

the B bus in the rest of the word). A conceptual picture of the shifter is shown in figure 3. 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 the A bus appears in the shift output. The shifter operates on $\varphi 1$; it may be viewed as an extension of the buses.

ALU

A block diagram of a single bit of the ALU is shown in figure 4. The ALU operates on the data which is contained in its two input latches. Input latch A may be loaded from the A bus, the shifter output, or the shift control, while the input latch B may be loaded from the B bus, the shifter output, or the shift control.

The outputs of the two latches become the inputs to two function blocks which 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 are 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 = (KFF, 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 the programming section.

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.

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 two-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 which can also be a source or destination of the A bus. 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 subtaction. 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 was 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.

Literal

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

Programming

The Datachip 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 shows the arrangement of the microcode word.

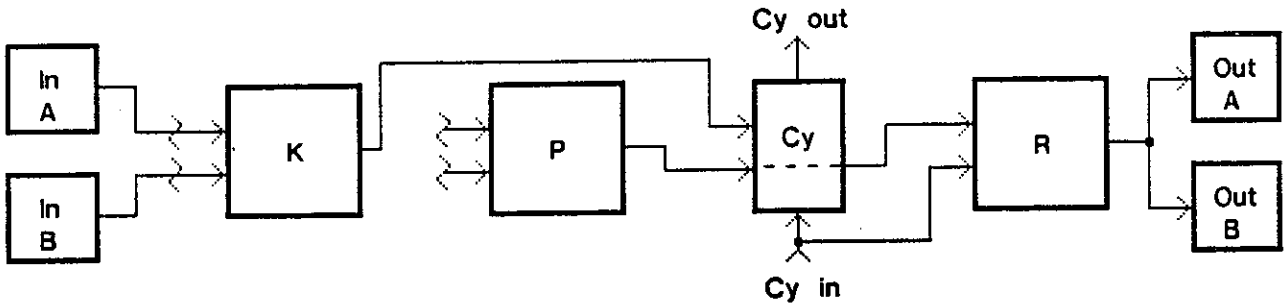


Figure 4. Block Diagram of one bit of the ALU

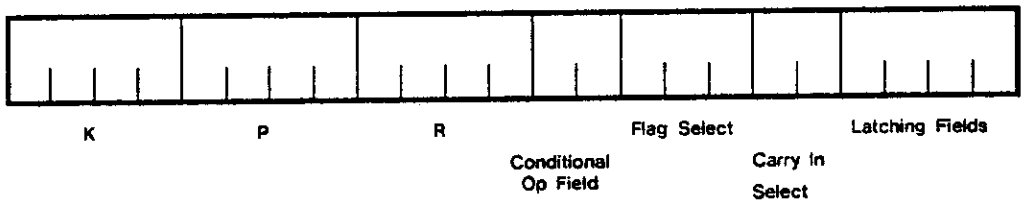


Figure 5a. Phi 2 Op Code (in on Phi 1)

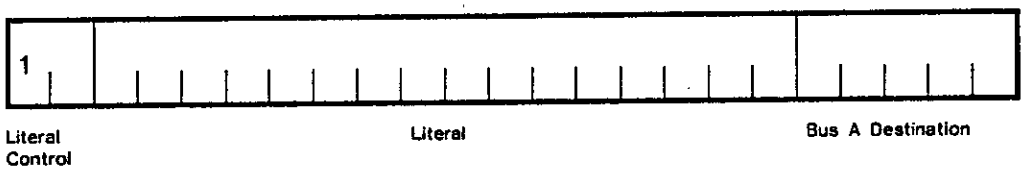


Figure 5b. Phi 1 Literal Transfer Op Code (in on Phi 2)

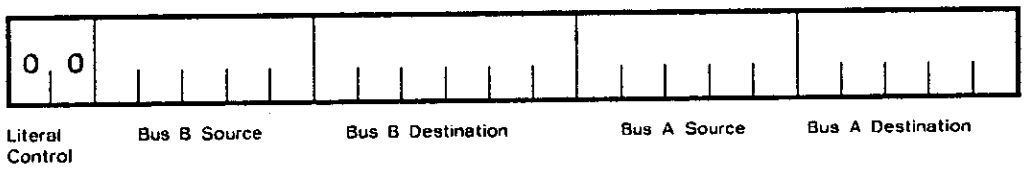


Figure 5c. Phi 1 Normal Op Code (in on Phi 2)

Bus Transfer

The bus transfer control bits enter the datachip during $\varphi 2$ and are used during the following $\varphi 1$. There are two buses, the A bus and the B bus, which interconnect the modules of the Datachip. 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 which is active is the A bus destination, which stores the literal entered on the A bus. A listing of bus sources and destinations follows:

	A Bus Source
0nnnn	Register n
10000	Right port pins
10001	Right port latch
10010	Left port pins
10011	Left port latch
10100	ALU output latch A
10101	ALU output latch B
10110	Flag register

	A Bus Destination
0nnnn	Register n
10000	Left port, drive now
10001	Left port, drive $\varphi 2$
1001x	Left port, no drive
10100	Right port, drive now
10101	Right port, drive $\varphi 2$
1011x	Right port, no drive
11000	ALU input latch A
11001	ALU input latch A gets shift out
11010	ALU input latch A gets shift ctl.
11011	Flag Register

	B Bus Source
0nnnn	Register n
10000	Right port pins
10001	Right port latch
10010	Left port pins
10011	Left port latch
10100	ALU output latch A
10101	ALU output latch B

	B Bus Destination
00nnnn	Register n
010000	Left port, drive now
010001	Left port, drive $\varphi 2$
01001x	Left port, no drive
010100	Right port, drive now
010101	Right port, drive $\varphi 2$
01011x	Right port, no drive
0110xx	ALU input latch B
10nnnn	ALU input latch B gets shift output, shift const. = n
11nnnn	ALU input latch B gets shift control, shift const. = n

ALU Input Selection

The two ALU input latches are destinations for the two buses, as shown in the Bus Transfer section 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 4 least significant bits of the B Bus Destination field, even though the destination

of the B Bus is not the ALU input latch B. For example, the B Bus may be transferring the contents of register 3 into register 5 while the A Bus 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 4 least significant bits of the B Bus Destination field contain "0101".

ALU Operations

The following table 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 cary in select, and **Cond** is the conditional OP select.

	K	**P**	**R**	**Cin**	**Cond**	
A+B	1	6	6	0	0	Add
A+B+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-Cin	2	9	6	1	0	Subtract with borrow
B-A-Cin	4	9	6	1	0	Subtract rev. w/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∧B	0	8	12	0	0	Logical And
A∨B	0	14	12	0	0	Logical Or
A⊕B	0	6	12	0	0	Logical Exor
¬A	0	3	12	0	0	Not A
¬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/O	0	14	12	0	3	Conditional And/Or
Mask	10	5	8	2	0	Generate mask

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 3 basic conditional type operations: Multiply, Divide, and And/Or step. In a great many cases, the conditional op allows functions dependant 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 selectand 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-	----	

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 bit
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 AL 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 $\varphi 2$ is loaded with the contents of the A Bus during the last $\varphi 2$
0000	None of these registers are affected

Literals

The two 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 transfered. If the second bit is 1, the literal goes off chip, while if the bit is 0, the literal comes on chip.

Programming Examples

This section of the memo contains 3 programming examples which should provide a better understanding of the various datapaths 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 X 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 6 illustrates each step of the loop, which is listed here:

```
φ2:  ALU.Out.A←ALU(Shift left)←ALU.In.A;
     Latch Flags;
φ1:  ALU.In.A←Shift.out, Bus.A←ALU.Out.B;
     Bus.B←R[1];          ←This gives a shift constant of 1.
φ2:  ALU.Out.B←ALU(Multiply Step);     ←conditionally add.
     Flag←Cout;
φ1:  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 the B input rotated by 1, 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 ⊕ bit *i-1* of the same input. If this same operation is performed, this time rotating the B input by 2, bit *i* becomes *i* ⊕ *i-1* ⊕ *i-2* ⊕ *i-3*. By doing this 2 more times, rotating B first by 4 and then by 8, every bit of the output is equal to the parity: 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 here, and illustrated in figure 7:

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

The third example adds all of the registers to what is in ALU.Out.A. By executing and modifying a literal, the registers can be indirectly accessed, which makes this routine possible. Figure 8 illustrates the operation of the following code:

```
φ1:  ALU.In.A←Literal "Bus.A←R[1]; ALU.In.B←Bus.B←ALU.Out.B";
φ2:  ALU.Out.B←ALU←ALU.In.A;
φ1:  ALU.In.A←Bus.A←R[0];
φ2:  ALU.Out.B←ALU←ALU.In.A;        ←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)←ALU.In.B;        ←point to next register.
```

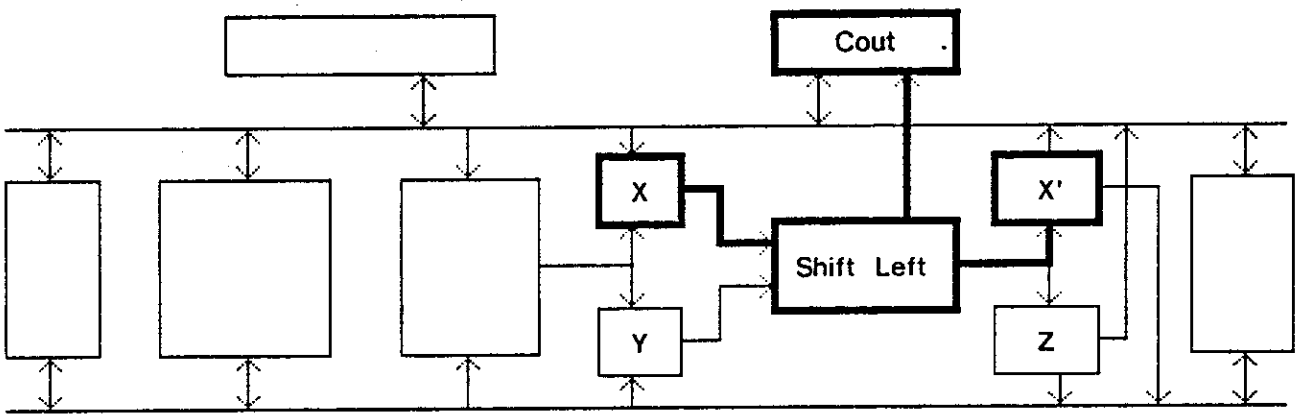


Figure 6a. Shift X in the ALU, putting the Cout flag into Flagbit. (Phi 2)

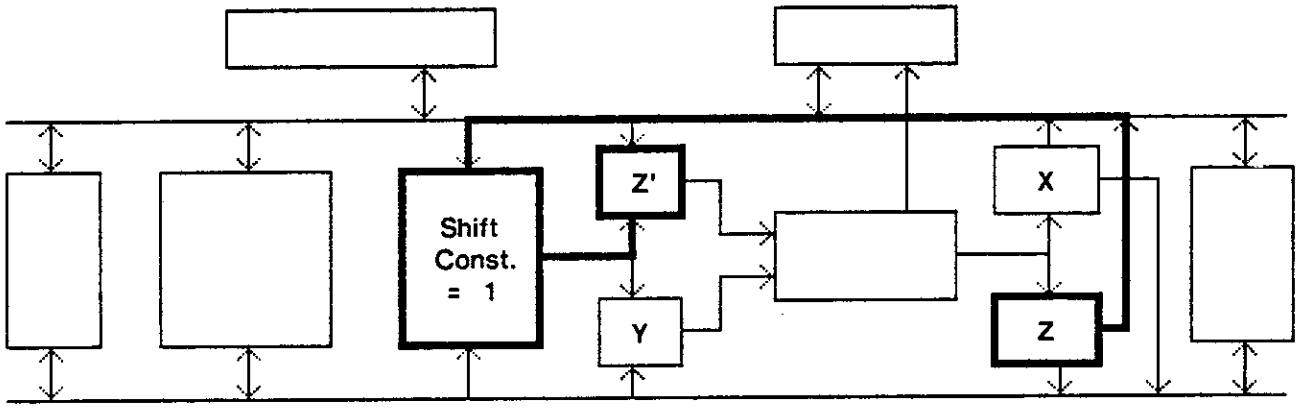


Figure 6b. Put Z on Bus A, and shift 1 left in shifter. (Phi 1)

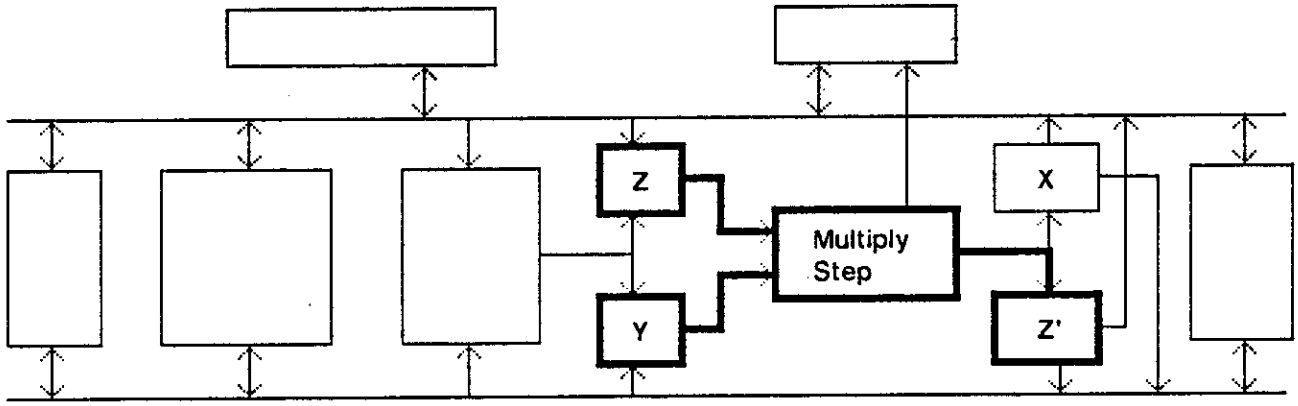


Figure 6c. Conditionally add Z and Y. (Phi 2)

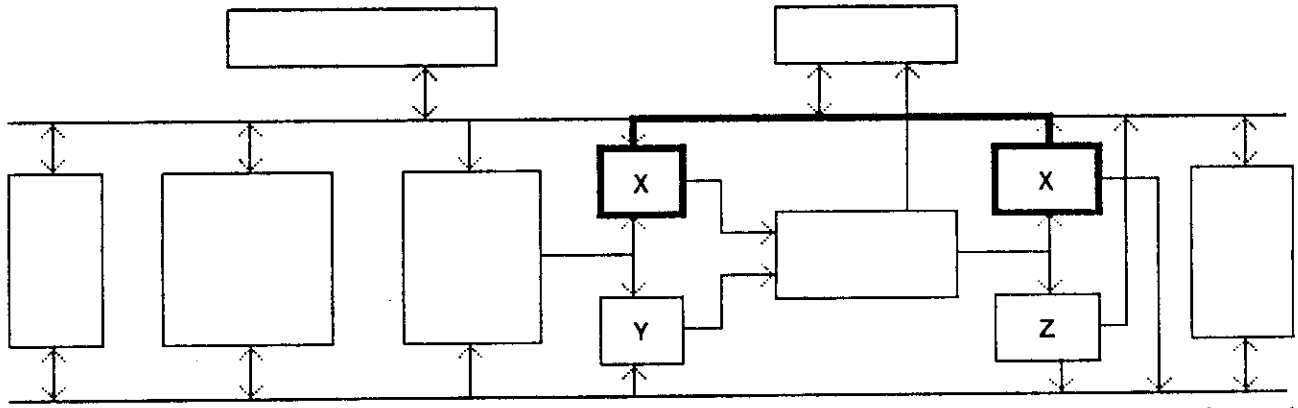


Figure 6d. Bring X back around to the ALU input. (Phi 1)

(fig6.sil)

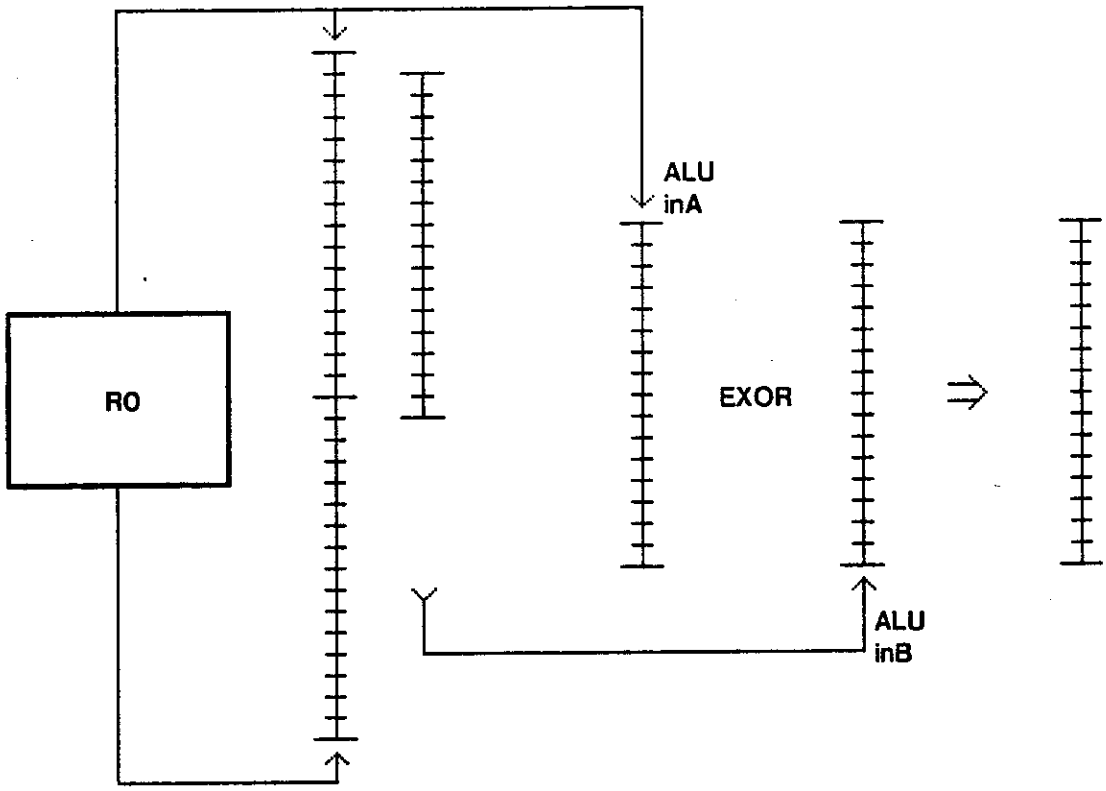


Figure 7a. Shifting by 1: Result is Exclusive-Or of Adjacent Bits.

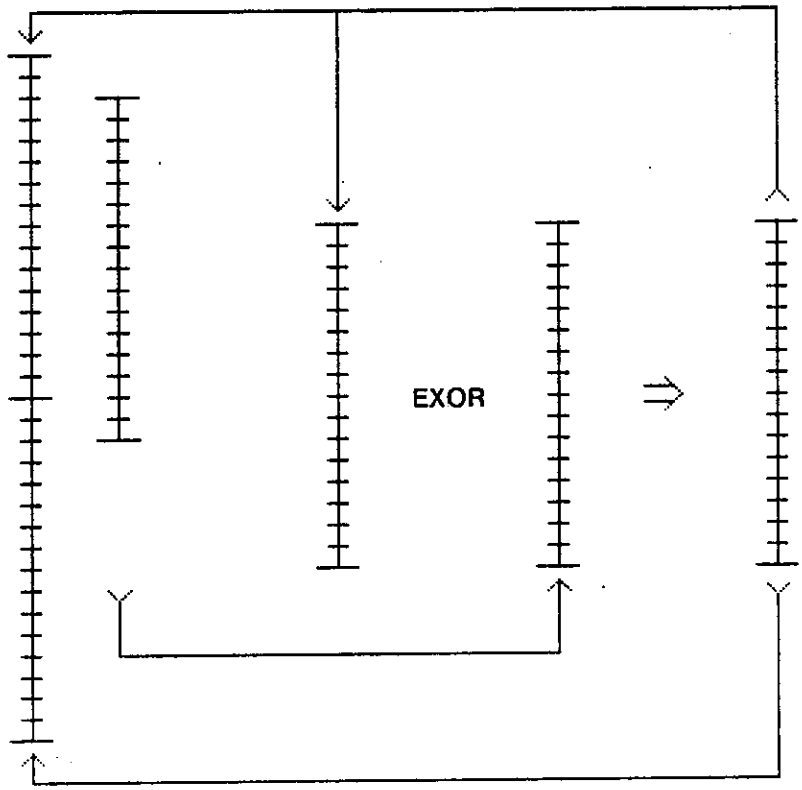


Figure 7b. Shifting by 2: Result is Exclusive-Or of 4 Adjacent Bits

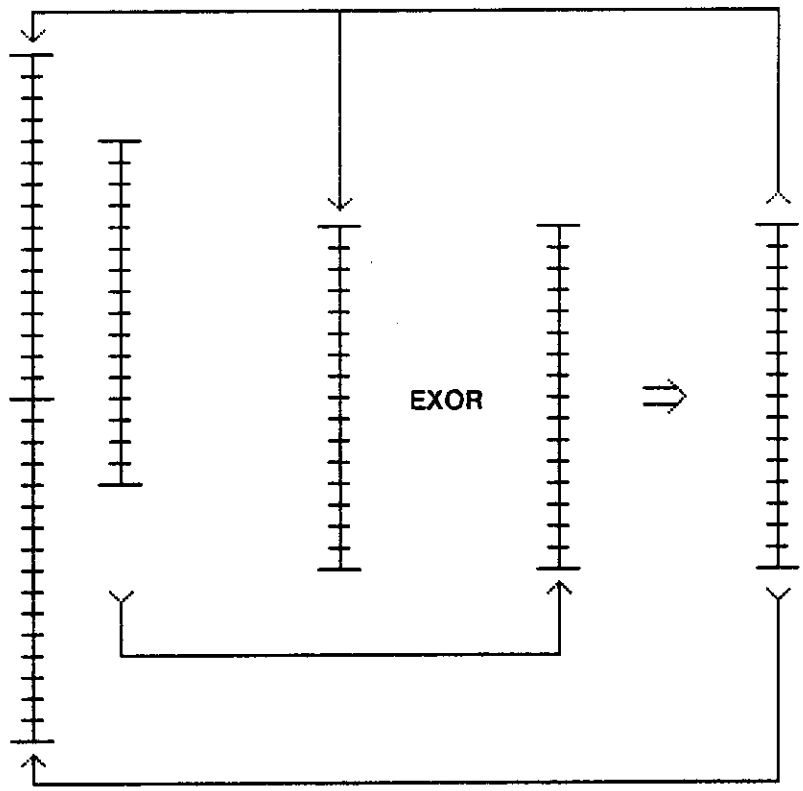


Figure 7c. Shifting by 4: Result is Exclusive-Or of 8 Adjacent Bits.

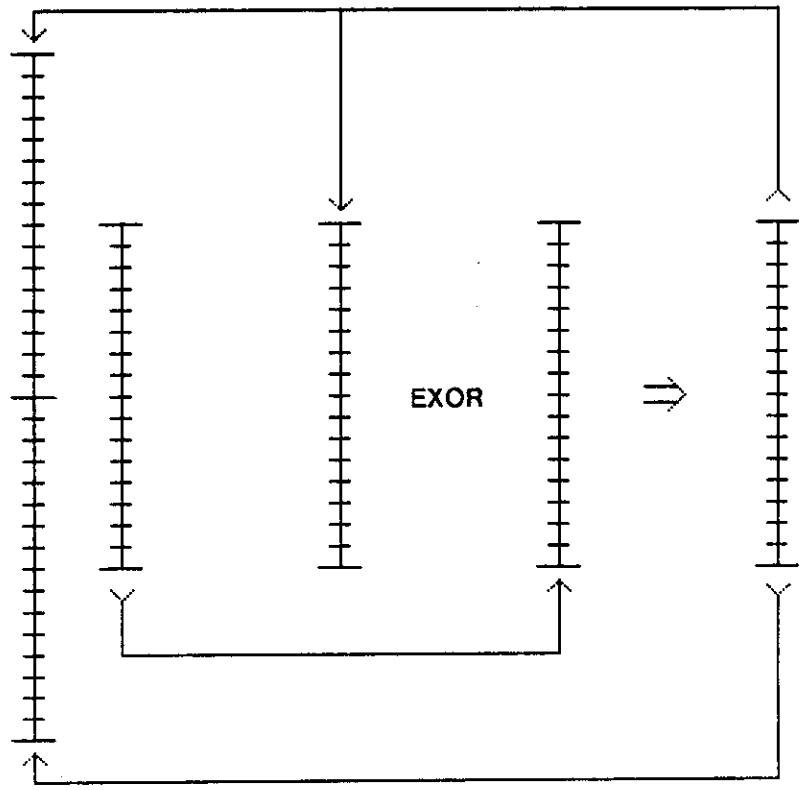


Figure 7d. Shifting by 8. Result Has All Bits Identically the Parity Flag.

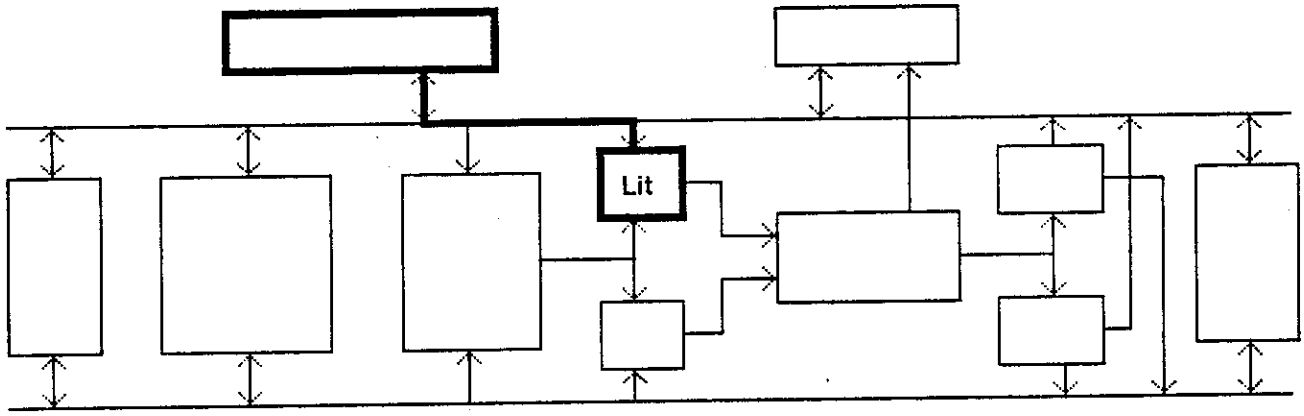


Figure 8a. Bring in Control Literal

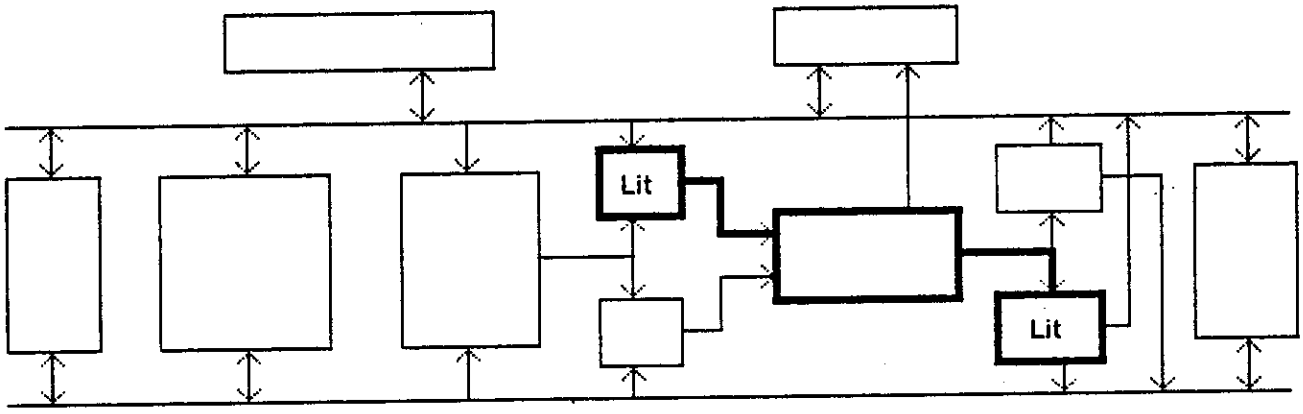


Figure 8b. Store in ALU.Out.B

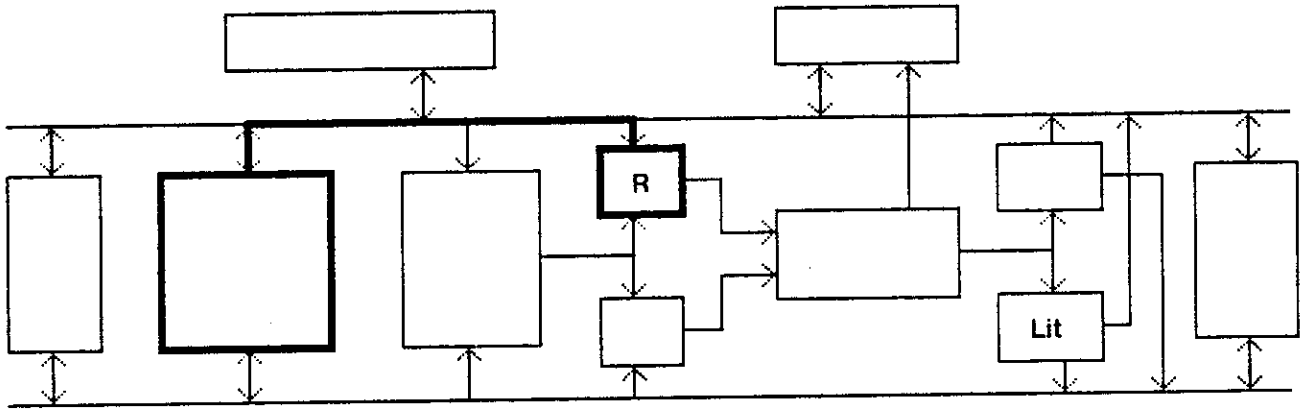


Figure 8c. Fetch Register 0

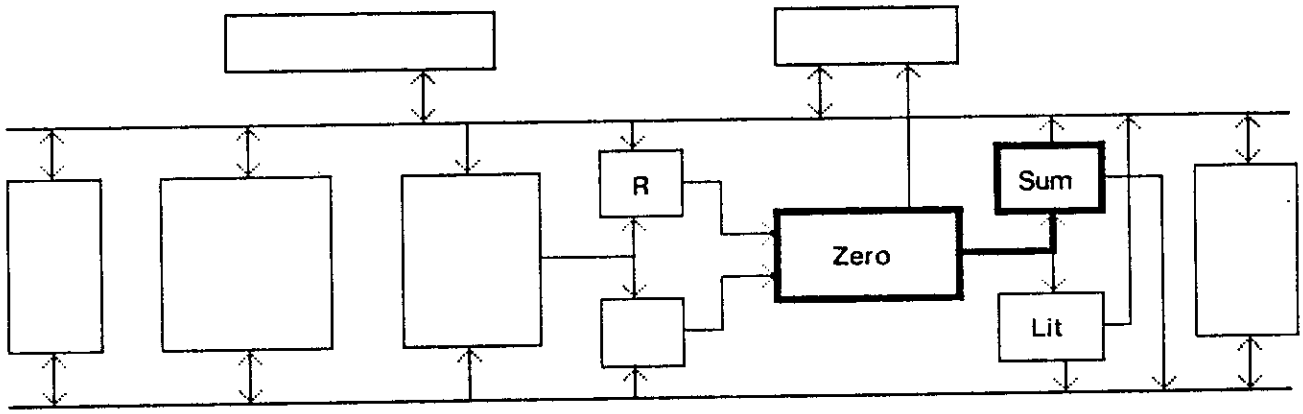


Figure 8d. Clear Sum

(fig8abcd.sil)

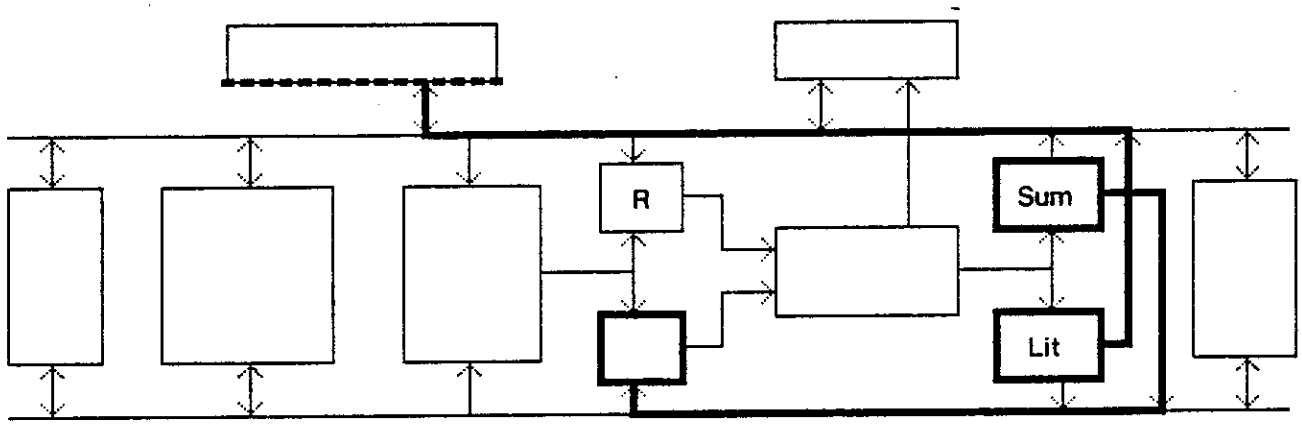


Figure 8e. Bring Around Sum and Put Control Literal on Bus A

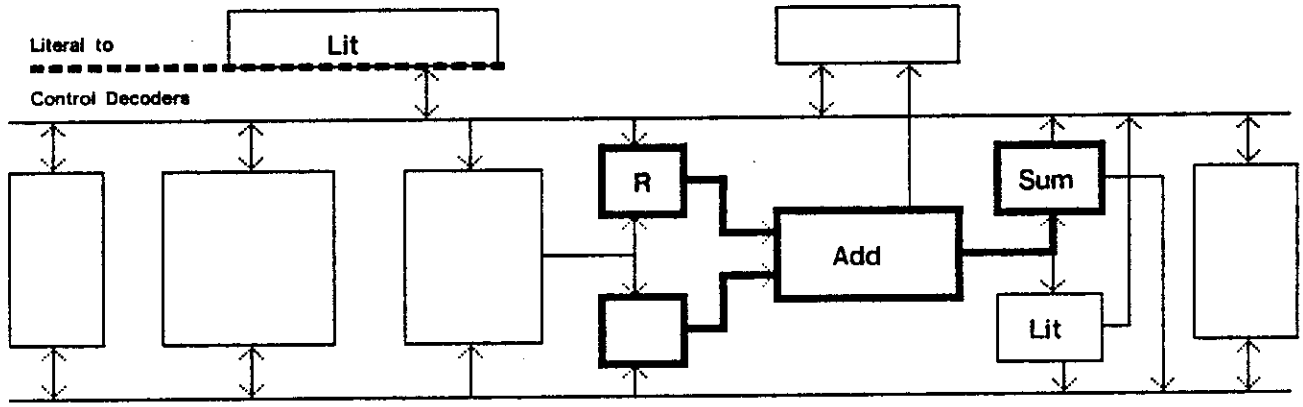


Figure 8f. Add Current Numbers

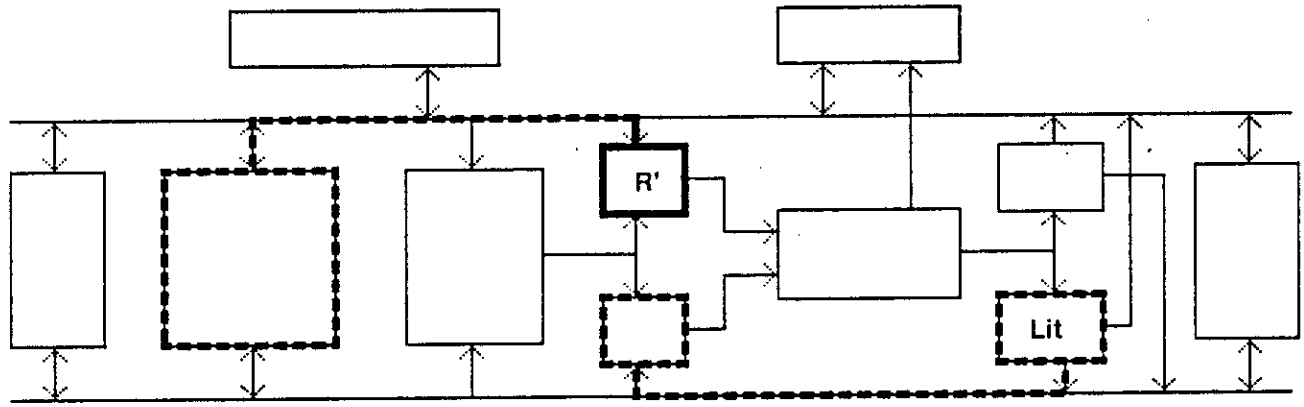


Figure 8g. Register Loaded by Literal Goes to ALU Input A

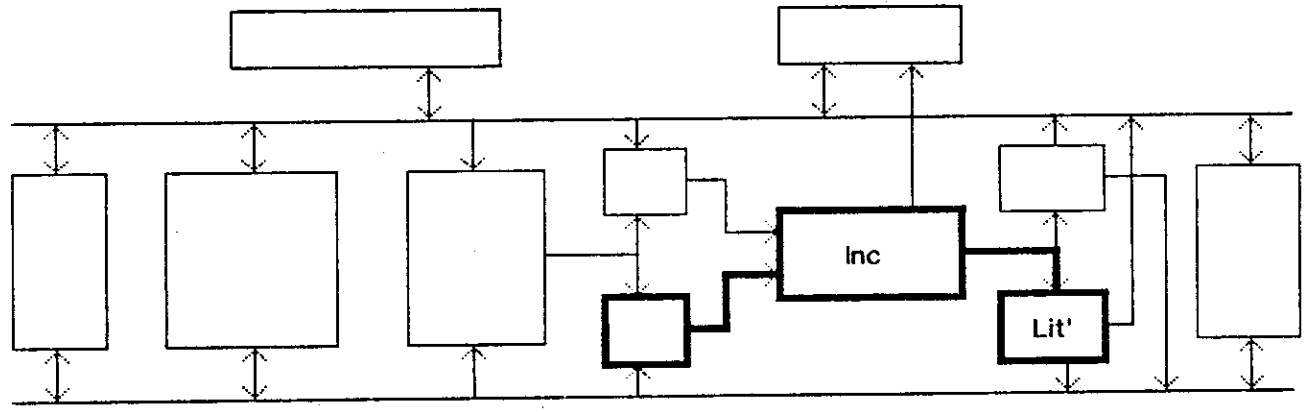


Figure 8h. Point to Next Register, Loop to Figure 8e (fig8efgh.sil)

ISP Description of the OM2 Datachip[2]

```
Pin States
   lp < 0:17 >                          left port
   rp < 0:17 >                          right port
   new.code < 0:22 >                    microcode
   flag.pin < 0 >                       flag to controller
   power < 0:3 >                        power, ground, clock, substrate
Pin Formats
   left.port.data < 0:15 >      := lp < 0:15 >
   left.out.async < 0 >         := lp < 16 >
   left.in.async < 0 >          := lp < 17 >
   right.port.data < 0:15 >     := rp < 0:15 >
   right.out.async < 0 >        := rp < 16 >
   right.in.async < 0 >         := rp < 17 >
   literal < 0:15 >             := new.code < 5:20 >
   clock < 0 >                  := power < 3 >
Mp State
   reg[0:15] < 0:15 >                   registers
   a.bus < 0:15 >                       bus a
   a.bus.old < 0:15 >                   bus a latched for a literal
   b.bus < 0:15 >                       bus b
   left.out < 0:15 >                    left pad output latch
   left.in < 0:15 >                     left pad input latch
   right.out < 0:15 >                   right pad output latch
   right.in < 0:15 >                    right pad input latch
   left.out.later < 0 >                 for output during φ2 operations
   right.out.later < 0 >                for output during φ2 for right port
   alu.in.a < 0:15 >                    alu input latch a
   alu.in.b < 0:15 >                    alu input latch b
   alu.out.a < 0:15 >                   alu output latch a
   alu.out.b < 0:15 >                   alu output latch b
   old.code < 0:22 >                    microcode that came in last phase
   flags < 0:15 >                       flag register
Instruction format
   a.source < 0:4 >             := old.code < 5:9 >
   b.source < 0:4 >             := old.code < 16:20 >
   a.destination < 0:4 >        := old.code < 0:4 >
   b.destination < 0:5 >        := old.code < 10:15 >
   literal.in < 0 >             := old.code < 22 >
   old.literal < 0:15 >         := old.code < 5:20 >
   alu.p.op < 0:3 >             := old.code < 19:22 >
   alu.k.op < 0:3 >             := old.code < 15:18 >
   alu.r.op < 0:3 >             := old.code < 11:14 >
   alu.conditional < 0:1 >      := old.code < 9:10 >
   flag.select < 0:2 >          := new.code < 6:8 >
   carry.in.select < 0:1 >      := old.code < 4:5 >
   latch.flags < 0 >            := old.code < 3 >
   latch.alu.out.a < 0 >        := old.code < 2 >
   latch.alu.out.b < 0 >        := old.code < 1 >
   literal.control < 0 >        := old.code < 0 >
   reg.select.1 < 0:3 >         := a.source < 0:3 >
   reg.select.2 < 0:3 >         := a.destination < 0:3 >
   reg.select.3 < 0:3 >         := b.source < 0:3 >
```

```
    reg.select.4 < 0:3 >                    := b.destination < 0:3 >
    select.1 < 0 >                          := a.source < 4 >
    select.2 < 0 >                          := a.destination < 4 >
    select.3 < 0 >                          := b.source < 4 >
    select.4 < 0:1 >                        := b.destination < 4:5 >
    shift.constant < 0:3 >                  := b.destination < 0:3 >
    sharay < 0:31 >                         := b.bus < 0:15 > □a.bus < 0:15 >

Temporary State
    kill.control < 0:3 >
    propagate.control < 0:3 >
    result.control < 0:3 >
    kill < 0:15 >
    propagate < 0:15 >
    carry < 0:16 >
    alu.out < 0:15 >

Instruction Execution
Instruction.execution: = (
    left.out.async = 0⇒(left.port.data←left.out);next
    left.in.async = 0⇒(left.in←left.port.data);next
    right.out.async = 0⇒(right.port.data←right.out);next
    right.in.async = 0⇒(right.in←right.port.data);next
    phi1(: = clock = 1)⇒(
        left.out.later←0;next
        right.out.later←0;next
        literal.in = 1⇒(a.bus←old.literal);next
        literal.in = 0⇒(
            select.1 = 0⇒(a.bus←reg[reg.select.1]);
            select.1 = 1⇒(
                reg.select.1 = 0⇒(a.bus←right.in←right.port.data);
                reg.select.1 = 1⇒(a.bus←right.in);
                reg.select.1 = 2⇒(a.bus←left.in←left.port.data);
                reg.select.1 = 3⇒(a.bus←left.in);
                reg.select.1 = 4⇒(a.bus←alu.out.a);
                reg.select.1 = 5⇒(a.bus←alu.out.b);
                reg.select.1 = 6⇒(a.bus←flags);next);next
            select.3 = 0⇒(b.bus←reg[reg.select.3]);
            select.3 = 1⇒(
                reg.select.3 = 0⇒(b.bus←right.in←right.port.data);
                reg.select.3 = 1⇒(b.bus←right.in);
                reg.select.3 = 2⇒(b.bus←left.in←left.port.data);
                reg.select.3 = 3⇒(b.bus←left.in);
                reg.select.3 = 4⇒(b.bus←alu.out.a);
                reg.select.3 = 5⇒(b.bus←alu.out.b);next);next
            select.4 = 0⇒(reg[reg.select.4]←b.bus);
            select.4 = 1⇒(
                reg.select.4 = 0⇒(left.port.data←left.out←b.bus);
                reg.select.4 = 1⇒(
                    left.out←b.bus;next
                    left.out.later←1;next);
                reg.select.4 = 2⇒(left.out←b.bus);
                reg.select.4 = 3⇒(left.out←b.bus);
                reg.select.4 = 4⇒(right.port.data←right.out←b.bus);
```

```
                    reg.select.4 = 5⇒(
                        right.out←b.bus;next
                        right.out.later←1;next);
                    reg.select.4 = 6⇒(right.out←b.bus);
                    reg.select.4 = 7⇒(right.out←b.bus);
                    reg.select.4∈{8,9,10,11}⇒(alu.in.b←b.bus);next);
                select.4 = 2⇒(alu.in.b < 0:15 > ←sharay < 16-shift.constant:31-shift.constant > );
                select.4 = 3⇒(alu.in.b←2↑shift.constant);next);next
            select.2 = 0⇒(reg[reg.select.2]←a.bus);
            select.2 = 1⇒(
                reg.select.2 = 0⇒(left.port.data←left.out←a.bus);
                reg.select.2 = 1⇒(
                    left.out←a.bus;next
                    left.out.later←1;next);
                reg.select.2 = 2⇒(left.out←a.bus);
                reg.select.2 = 3⇒(left.out←a.bus);
                reg.select.2 = 4⇒(right.port.data←right.out←a.bus);
                reg.select.2 = 5⇒(
                    right.out←a.bus;next
                    right.out.later←1;next);
                reg.select.2 = 6⇒(right.out←a.bus);
                reg.select.2 = 7⇒(right.out←a.bus);
                reg.select.2 = 8⇒(alu.in.a←a.bus);
                reg.select.2 = 9⇒(alu.in.a < 0:15 > ←sharay < 16-shift.constant:31-shift.constant > );
                reg.select.2 = 10⇒(alu.in.a←2↑shift.constant);
                reg.select.2 = 11⇒(flags←a.bus);next);next
            flag.select = 1⇒(flags < 15 > ←flags < 14 > );
            flag.select = 2⇒(flags < 15 > ←flags < 12 > );
            flag.select = 3⇒(flags < 15 > ←flags < 11 > );
            flag.select = 4⇒(flags < 15 > ←flags < 9 > );
            flag.select = 5⇒(flags < 15 > ←flags < 7 > );
            flag.select = 6⇒(flags < 15 > ←flags < 8 > );
            flag.select = 7⇒(flags < 15 > ←flags < 13 > );next

phi2(: = clock = 0)⇒(
    left.out.later = 1⇒(left.port.data←left.out);next
    right.out.later = 1⇒(right.port.data←right.out);next
    kill.control←alu.k.op;next
    propagate.control←alu.p.op;next
    result.control←alu.r.op;next
    alu.conditional = 1⇒(
        flags < 15 > = 1⇒(
            propagate.control < 0 > ←0;next
            result.control < 0 > ←0;next);
        flags < 15 > = 0⇒(
            kill.control < 3 > ←0;next
            propagate.control < 2 > ←0;next
            result.control < 2 > ←0;next);next);
    alu.conditional = 2⇒(
        flags < 15 > = 1⇒(
            kill.control < 2 > ←0;next
            kill.control < 1 > ←0;next
            propagate.control < 3 > ←0;next
            propagate.control < 0 > ←0;next
            result.control < 3 > ←0;next
            result.control < 0 > ←0;next);
```

```
        flags < 15 > = 0⇒(
            kill.control < 3 > ←0;next
            kill.control < 0 > ←0;next
            propagate.control < 2 > ←0;next
            propagate.control < 1 > ←0;next
            result.control < 2 > ←0;next
            result.control < 1 > ←0;next);next);
    alu.conditional = 3⇒(
        flags < 15 > = 1⇒(
            propagate.control < 2 > ←0;next
            propagate.control < 1 > ←0;next);next);next
    kill < 0:15 > ←(
        kill.control < 3 > ∧(¬alu.in.a < 0:15 > )∧(¬alu.in.b < 0:15 > )∨
        kill.control < 2 > ∧(¬alu.in.a < 0:15 > )∧alu.in.b < 0:15 > ∨
        kill.control < 1 > ∧alu.in.a < 0:15 > ∧(¬alu.in.b < 0:15 > )∨
        kill.control < 0 > ∧alu.in.a < 0:15 > ∧alu.in.b < 0:15 > );next
    propagate < 0:15 > ←(
        propagate.control < 3 > ∧(¬alu.in.a < 0:15 > )∧(¬alu.in.b < 0:15 > )∨
        propagate.control < 2 > ∧(¬alu.in.a < 0:15 > )∧alu.in.b < 0:15 > ∨
        propagate.control < 1 > ∧alu.in.a < 0:15 > ∧(¬alu.in.b < 0:15 > )∨
        propagate.control < 0 > ∧alu.in.a < 0:15 > ∧alu.in.b < 0:15 > );next
    carry < 0 > ←carry.in.select < 1 > ⊕(carry.in.select < 0 > ∧flags < 15 > );next
    for k = 1 step 1 until 16 do:
        (carry < k > ←¬(kill < k-1 > + propagate < k-1 > *¬carry < k-1 > ) + kill < k-1 > *
                propagate < k-1 > *x);next                    in OM2, x is undefined
        If kill(i) and propagate(i) are both high, the carry chain does funny things.
        We represent that here by use of the "x" in the carry function.
    alu.out < 0:15 > ←(
        result.control < 3 > ∧(¬propagate < 0:15 > )∧(¬carry < 0:15 > )∨
        result.control < 2 > ∧(¬propagate < 0:15 > )∧carry < 0:15 > ∨
        result.control < 1 > ∧propagate < 0:15 > ∧(¬carry < 0:15 > )∨
        result.control < 0 > ∧propagate < 0:15 > ∧carry < 0:15 > );next
    latch.alu.out.a = 1⇒(alu.out.a←alu.out);next
    latch.alu.out.b = 1⇒(alu.out.b←alu.out);next
    literal.control = 1⇒(literal←bus.a.old);next
    latch.flags = 1⇒(
        flags < 5 > ←flags < 15 > ;next
        flags < 6 > ←carry < 15 > ;next
        flags < 10 > ←alu.out < 0 > ;next
        flags < 11 > ←0;next
        alu.out = 0⇒(flags < 11 > ←1);next
        flags < 12 > ←alu.out < 15 > ;next
        flags < 14 > ←carry < 16 > ;next
        flags < 13 > ←flags < 14 > ⊕flags < 6 > ;next
        flags < 9 > ←flags < 12 > ⊕flags < 13 > ;next
        flags < 7 > ←flags < 11 > ∨flags < 9 > ;next
        flags < 8 > ←¬(flags < 14 > ∨flags < 11 > );next);next);next
                    )                    end of instruction execution
```

References:

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

2. C. G. Bell, A. Newell, "Computer Structures: Readings and Examples", Chapter 2, McGraw-Hill, 1971.

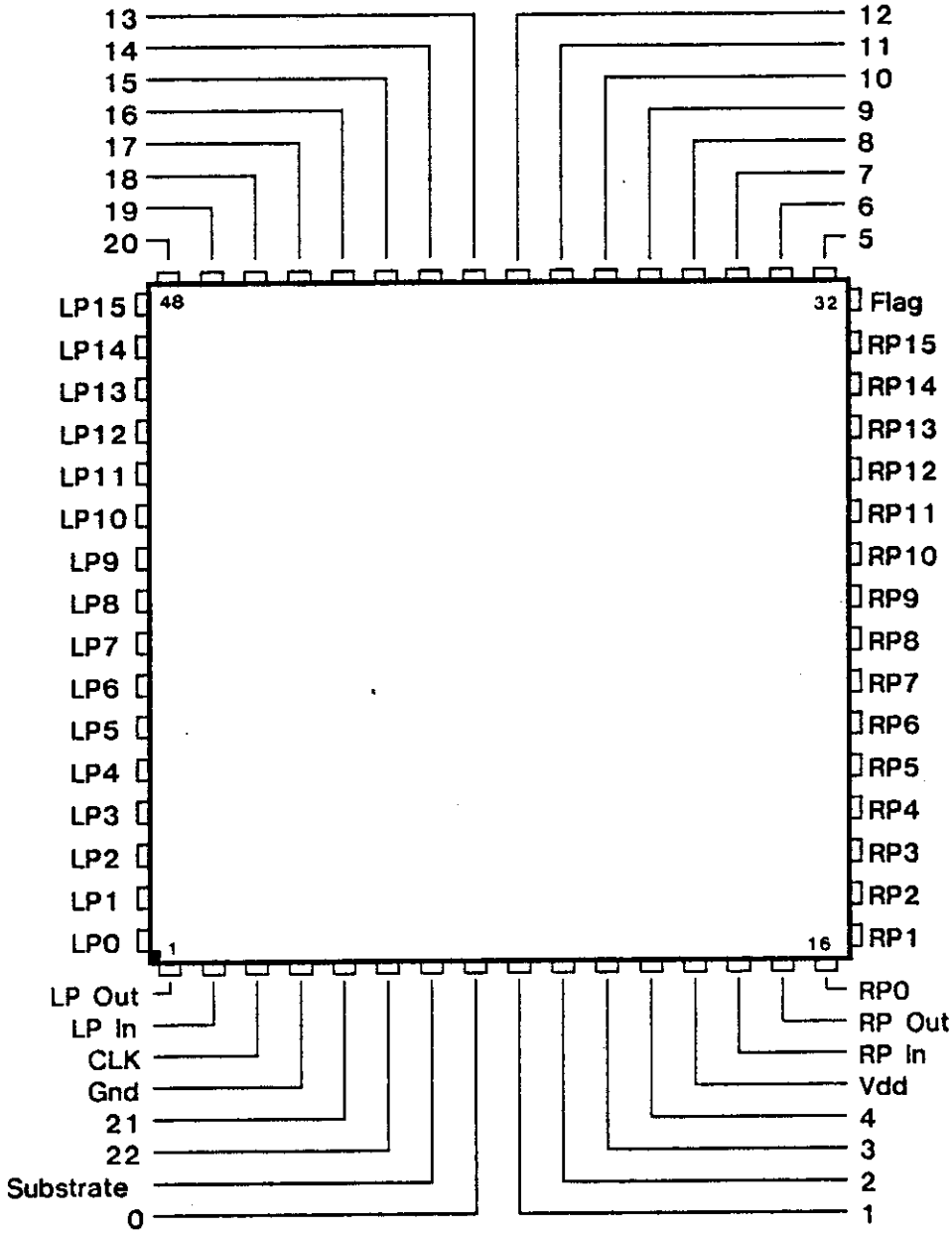
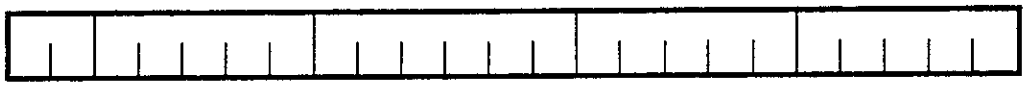


Figure 9. Pinout of the OM2 Datachip

Summary of Commands

Transfer Phase: PHI 1



Bus A Source

0nnnn	Register n
10000	Right Port Pins
10001	Right Port Latch
10010	Left Port Pins
10011	Left Port Latch
10100	ALU Output Latch A
10101	ALU Output Latch B
10110	Flag Register
-------	Literal (see Literal Control)
other	No Source

Literal Control

000	Microcode In
001	Illegal
010	Literal In
011	Illegal
100	Execute old A Bus
101	Illegal
110	A Bus gets old A Bus
111	Literal Out

└── LSB of the Latching Field during last PHI 2.

Bus B Source

0nnnn	Register n
10000	Right Port Pins
10001	Right Port Latch
10010	Left Port Pins
10011	Left Port Latch
10100	ALU Output Latch A
10101	ALU Output Latch B
other	No Source

Bus A Destination

0nnnn	Register n
10000	Left Port, drive now
10001	Left Port, drive PHI 2
1001x	Left Port, no drive
10100	Right Port, drive now
10101	Right Port, drive PHI 2
1011x	Right Port, no drive
11000	ALU Input Latch A
11001	ALU Input Latch A gets Shift Out
11010	ALU Input Latch A gets Shift Control
11011	Flag Register
other	Do Destination

Bus B Destination

00nnnn	Register n
010000	Left Port, drive now
010001	Left Port, drive PHI 2
01001x	Left Port, no drive
010100	Right Port, drive now
010101	Right Port, drive PHI 2
01011x	Right Port, no dirve
0110xx	ALU Input Latch B
0111xx	No Destination
10nnnn	ALU Input Latch B gets shift output, shift constant=n
11nnnn	ALU Input Latch B gets shift control, shift constant=n

Operation Phase: PHI 2

ALU Operation

1000	0110	0110	00	00	Add
1000	0110	0110	00	01	Add with Carry
0100	1001	0110	00	10	Subtract
0010	1001	0110	00	10	Subtract Reversed
0100	1001	0110	00	01	Subtract with Borrow
0010	1001	0110	00	01	Subtract Reversed with Borrow
0011	1100	0110	00	10	Negative A
0101	1010	0110	00	10	Negative B
1100	0011	0110	00	10	Increment A
1010	0101	0110	00	10	Increment B
0011	1100	1001	00	10	Decrement A
0101	1010	1001	00	10	Decrement B
0000	0001	0011	00	00	Logical AND
0000	0111	0011	00	00	Logical OR
0000	0110	0011	00	00	Logical Exclusive Or
0000	1100	0011	00	00	Not A
0000	1010	0011	00	00	Not B
0000	0011	0011	00	00	A
0000	0101	0011	00	00	B
1000	0111	0111	01	00	Multiply Step
1100	1111	1111	10	00	Divide Step
0000	0111	0011	11	00	Conditional AND/OR
0101	1010	0001	00	10	Generate Mack
uuuu	uuuu	uuuu	uu	uu	User Defined Op

└── Carry In Select Field

Carry In Select

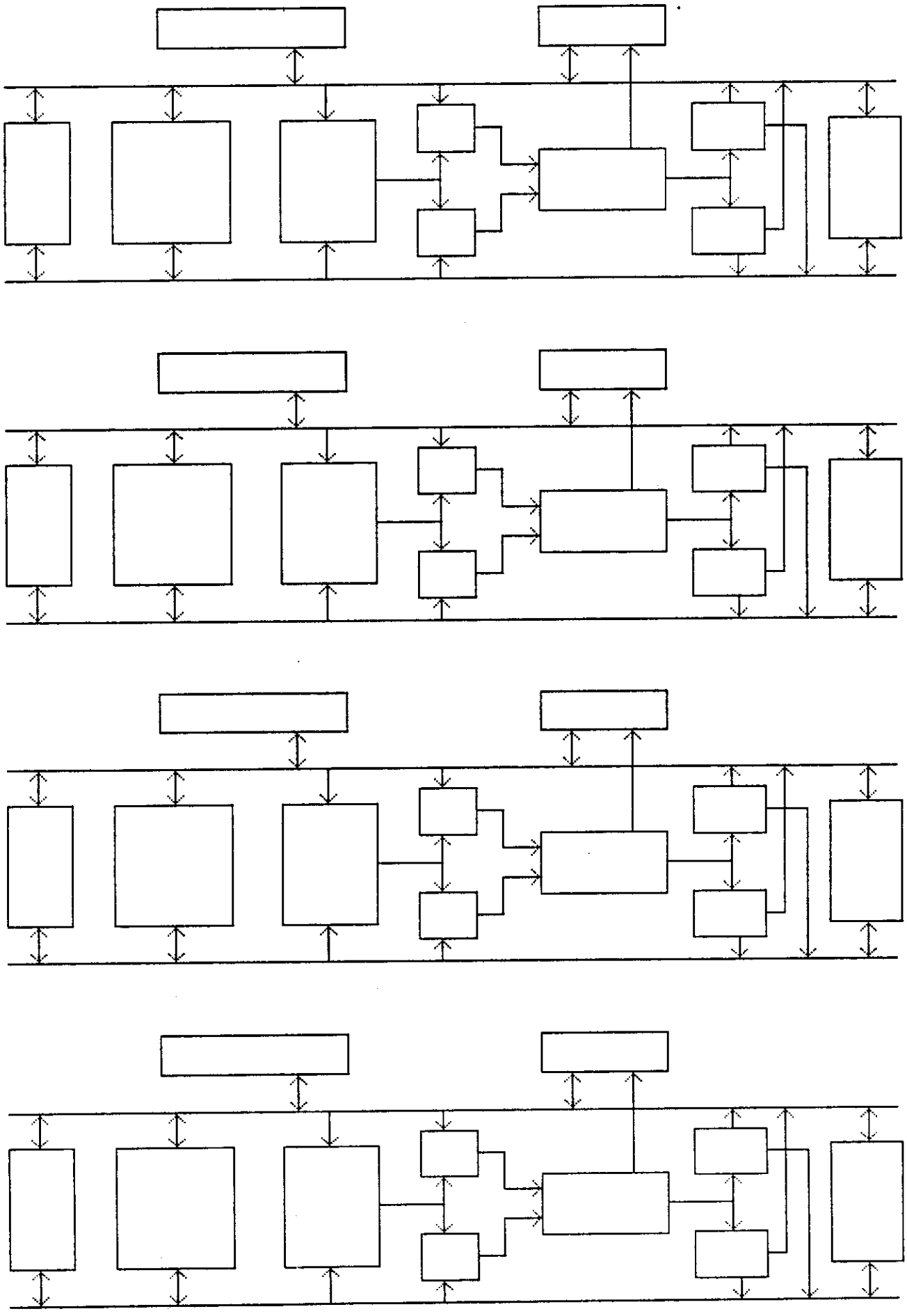
00	0
01	Flagbit
10	1
11	Flagbit Complimented

Flag Select

000	Old Flagbit
001	Carry Out
010	MSB
011	Zero
100	Less than flag
101	Less than or equal flag
110	Higher flag
111	Overflow

Latching Field

1xxx	Latch Flags
x1xx	Load ALU Output Latch A
xx1x	Load ALU Output Latch B
xxx1	Literal bits get old A Bus next PHI 1
0000	Nop



(fig11.sil)