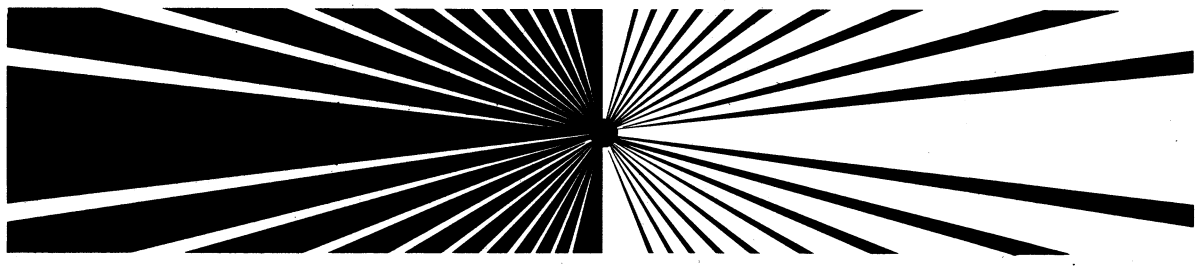
Structured VLSI design proceeds from algorithm to logic cell to cell array to special-purpose chip, yielding cheap, powerful, and modular hardware that will permanently alter the systems landscape of the 80's.



The Design of Special-Purpose VLSI Chips

M. J. Foster
H. T. Kung
Carnegie-Mellon University

We have now entered a technological domain in which many of the problems previously encountered in building special-purpose hardware are much less severe. LSI technology allows tens of thousands of devices to fit on a single chip, and the advance to VLSI should increase this number. Devices once requiring many components can now be built with just a few chips, reducing the difficulties in reliability, performance, and heat dissipation that arise from combining many standard SSI or MSI components. In addition, the development of simplified techniques[1] and implementation guides [2] for structuring IC system design—an area often regarded as difficult—allows relatively naive designers to achieve success.

Special-purpose VLSI chips can function as peripheral devices attached to a conventional host computer. If many types of chips are attached, the resulting system can be considered an efficient general-purpose computer. Figure 1 illustrates how special-purpose chips such as a pattern matcher, FFT device, and sorter might form part of such a general-purpose system.

Construction of complex special-purpose VLSI chips will be feasible only if we hold down design cost (i.e., design time). We will argue that chip design time can be reduced significantly if the underlying algorithm is "good"—i.e., designed carefully in the first place. We will characterize such algorithms below and will examine a concrete example of one—the design of a pattern matching chip. (We completed this design in the spring of 1979 and have had prototype chips fabricated, with testing now under way. Preliminary results show that the chip can achieve a data rate of one character every 250 nanoseconds, which is higher than the memory bandwidth of most conventional computers. This high performance is achieved in spite of little effort given to circuit and layout design. We attribute this performance mainly to the careful design of the underlying algorithm.)

We will also identify the major steps in special-purpose VLSI chip design according to the good algorithm philosophy, and will offer a methodology for transforming a good algorithm into a final layout in a more or less mechanical way. With this approach, designing a special-purpose chip should not be more difficult than designing a high-level algorithm for the same job.

Design philosophy

Algorithms that perform well on conventional random access computers are not always the best for VLSI implementation. As Sutherland and Mead[3] point out, good algorithms for VLSI implementation are not necessarily those requiring minimal computation. Computation is cheap in VLSI; *communication* determines performance. Thus, in this new era of computation, we need to reconsider the algorithms for many tasks.

A "good" algorithm in this context should possess one or more of the following properties:

- The algorithm can be implemented by only a *few* different types of *simple* cells.
- The algorithm's data and control flow is *simple* and *regular*, so that cells can be connected by a network with local and regular interconnections. Long distance or irregular communication is thus minimized.

0018-9162/80/0100-0026$00.75 © 1980 IEEE

- The algorithm uses extensive *pipelining* and *multiprocessing*. Typically, several data streams move at constant velocity over fixed paths in the network, interacting at cells where they meet. In this way a large number of cells are active at one time so that the computation speed can keep up with the data rate.

Algorithms with these properties have been named *systolic algorithms** by Kung and Leiserson.[4] Many have been designed recently and are surveyed by Kung.[5]

Since most special-purpose chips will be made in relatively small quantities, the design cost must be kept low. Systolic algorithms have several advantages which help reduce this cost:

- One can design and test only a few different, simple cells, since most of the cells on the chip are copies of a few basic ones.
- Regular interconnection implies that the design can be made modular and extensible, so one can design a large chip by combining the designs of small chips.
- By pipelining and multiprocessing, one can meet the performance requirement of a special-purpose chip simply by including many identical cells on the chip.

*The word "systole" was borrowed from physiologists, who use it to refer to the rhythmically recurrent contractions of the heart, which pulse blood through the body. For a systolic algorithm, the function of a cell is analogous to that of the heart. Each cell regularly pumps data in and out (performing some short computation before each "contraction"), so that a regular flow of data is kept up in the network.

All these imply that if a good algorithm is used, the design time, and therefore the design cost, can be substantially reduced.

In VLSI special-purpose chip design, then, the most crucial decision is the choice of the underlying algorithm, since the suitability of the algorithm largely determines the design cost and performance. Given the importance of algorithm design, it should receive the largest part of the design effort. Low-level optimizations at the circuit or layout design level are probably not worthwhile, as these will lead only to minor improvements in the overall performance while increasing design time.

The design of a pattern matching chip

A specific VLSI chip—one that performs on-line pattern matching of strings with wild card characters—illustrates our design philosophy and methodology. The design of the underlying algorithm demonstrates that it can be mapped to circuit and layout designs in a straightforward way.

The string pattern matching problem. Our chip accepts two streams of characters from the host machine, and produces a stream of bits as shown in Figure 2. One of the input streams, the text string, is an endless string of characters over some alphabet Σ. The other input stream, the pattern, contains a fixed-length vector of characters over the alphabet $\Sigma \cup \{X\}$, where X is the wild card character. The output is a stream of bits, each of which corresponds to one of the characters in the text string. The data streams move at a steady rate between the host computer and the

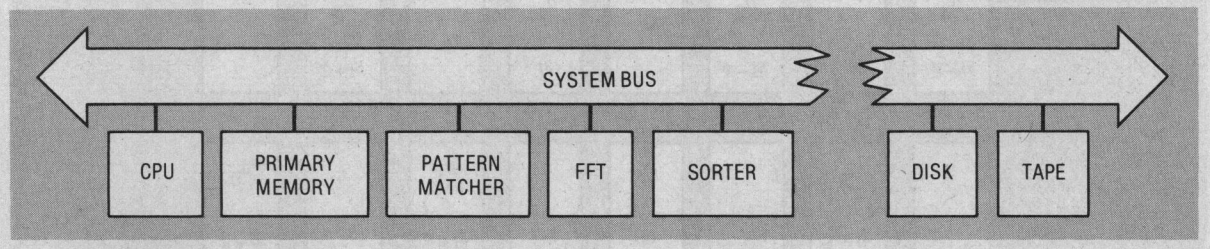


Figure 1. Special-purpose chips attached to a general-purpose computer.

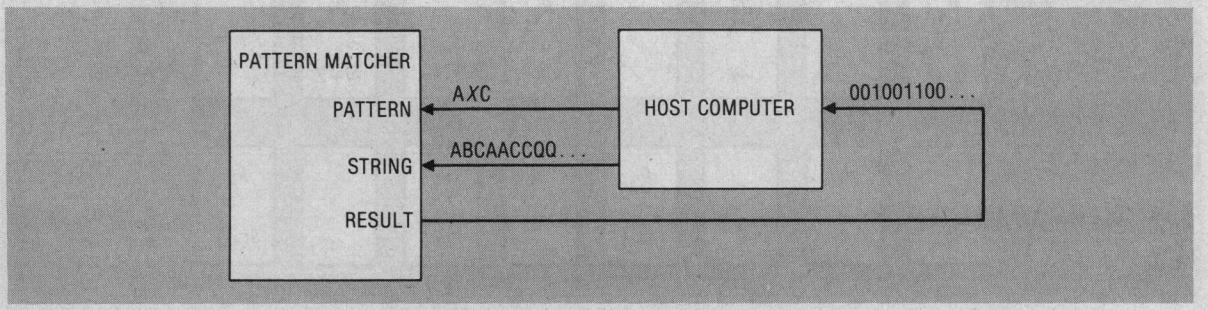


Figure 2. Data to and from the pattern matcher.

pattern matcher, with a constant time between data items.

Let us denote the input text stream as $s_0 s_1 s_2 \ldots$. The finite pattern stream will be denoted as $p_0 p_1 \ldots p_k$ and the output result stream as $r_0 r_1 r_2 \ldots$. Characters in the two input streams may be tested for equality, with the wild card character X deemed to match any character in Σ. The output bit r_i is to be set to 1 if the substring $s_{i-k} s_{i+1-k} \ldots s_i$ matches the pattern, and 0 otherwise, i.e.,

$$r_i \leftarrow (s_{i-k} = p_0) \wedge (s_{i+1-k} = p_1) \wedge \ldots \wedge (s_i = p_k).$$

In Figure 2, for example, the pattern AXC matches the substrings $s_0 s_1 s_2$, $s_3 s_4 s_5$, and $s_4 s_5 s_6$ (ABC, AAC, and ACC). Result bits $r_2 r_5$, and r_6 are thus set to 1, and all other result bits are 0.

This problem is important in many applications. String pattern matching is a basic operation in SNOBOL-like languages[6] and in data base query languages. String matching hardware has been proposed for use in office automation systems.[7] Many artificial intelligence systems make heavy use of pattern matching as a search method. Furthermore, string pattern matching is similar to many stressing numerical computations such as convolutions and correlations. All of the linear product problems discussed by Fischer and Paterson[8] are also similar to string matching.

Several fast algorithms are known for solving the string matching problem without wild card characters on a normal random access machine.[9, 10] These methods use information about partial matches of the pattern with itself to avoid redundant comparisons, skipping over parts of the string where partial match results may be inferred from previous comparisons. When wild card characters exist in the pattern these methods break down, since the "matches" relation is no longer transitive. The strings AC and XB both match AX, for example, but do not match each other. Information about matchings of the pattern with itself is therefore irrelevant if wild card characters are present. The fastest algorithm known for string matching with wild card characters is based on multiplication of large integers[8] and requires more than linear time. The pattern matching chip solves the problem in linear time by performing comparisons in parallel.

The chip design. We designed our chip according to the methodology discussed earlier, beginning with the careful design of a systolic algorithm and proceeding to its hardware implementation.

Algorithm design—data flow. The pattern and the text string arrive alternately over the bus one character at a time. We will call the interval during which one character arrives from either stream a beat. During each pair of consecutive beats the chip must input

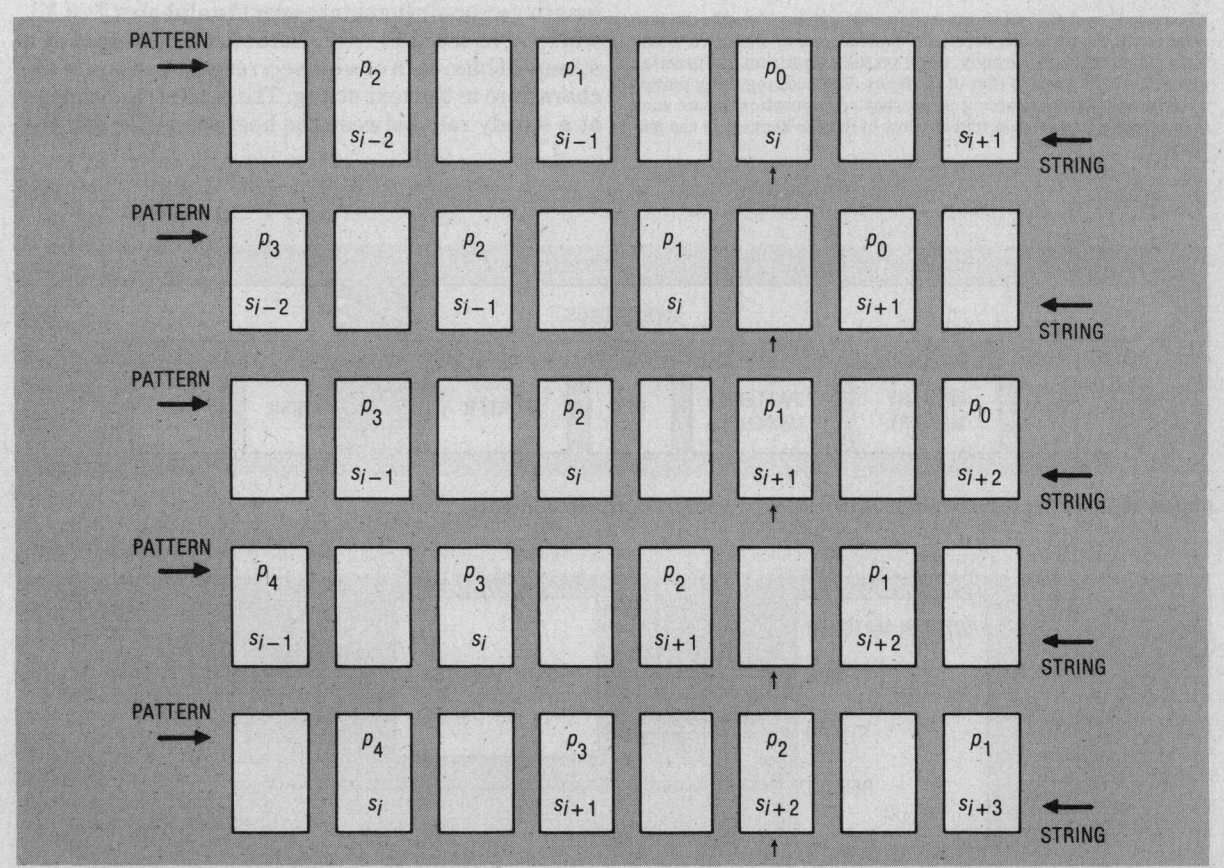


Figure 3. The flow of characters through a linear array of cells.

two characters and output one result. All characters on the chip move during each beat.

The chip is divided into character cells, each of which can compare two characters and accumulate a temporary result. The pattern and string follow a preset path of cells from the time they enter the chip until the time they leave it. On each beat every character moves to a new cell. We use a linear array of cells, with the pattern and string moving in opposite directions, to make each character of the string move past all characters of the pattern. To make each pair of characters meet rather than just pass, we must separate them by one cell so that alternate cells are idle. Each cell is then active on alternate beats. Figure 3 traces the flow of characters for several beats.

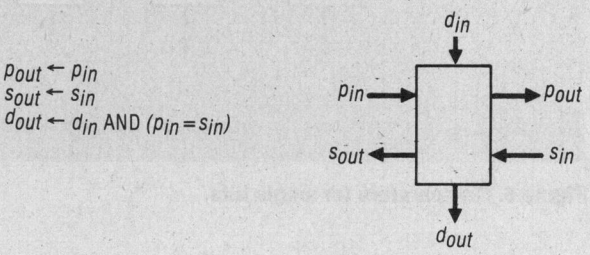
Following the pointer in Figure 3 illustrates the history of the character cell, starting when the first character of the pattern, p_0, is present. Suppose the string character s_i is present during this beat. During the next beat the cell is idle, but during the beat after that it contains p_1 and s_{i+1}. Two beats later, p_2 and s_{i+2} are together, then p_3 and s_{i+3}, and so on. By the time the last pattern character p_k leaves the cell, the substring $s_i s_{i+1} \ldots s_{i+k}$ will have met the whole pattern. We can therefore keep the partial match results in this cell, update it whenever a new pair of characters enters the cell, and output the results after the last character of the pattern goes past. To output results we shift them along with the string, so that each match result leaves the array with the last character of its substring. If we recirculate the pattern so that the first character follows two beats after the last one, we can output the completed result and initialize a new partial result on the same beat. The number of character cells required is therefore no more than the number of characters in the pattern.

Each character cell performs two separate functions—it compares characters of the pattern and string, and it updates and outputs match results. We can divide these functions between two modules, so that there are two linear arrays with connections between corresponding cells as shown in Figure 4. The cells on the top are the comparators; the pattern flows through them from left to right, the string from right to left. The bottom cells, or accumulators, receive the results of the comparison from above. They maintain partial results and shift completed results right to left. Two bits associated with the pattern flow through the accumulators from left to right. One of these bits, called λ, marks the end of the pattern. It is one for the last character of the pattern and zero for the others. The other bit is x, the "don't care" bit, which marks wild card characters. A one in this bit tells the accumulator to ignore the result from the comparator, since this pattern character matches anything.

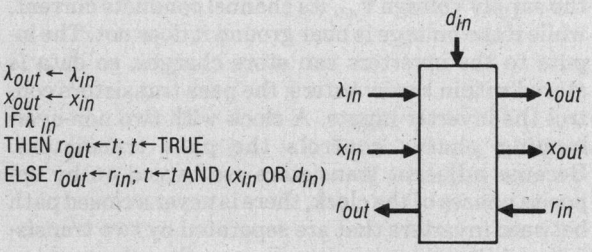
We can further divide the comparators. Rather than using one large circuit to compare whole characters, we can divide each comparator into modules that can compare single bits. Two characters are equal if corresponding bits are equal. By staggering the bits so the high-order bits enter the array before the low-order ones, we can make a pipeline comparator. Each single-bit comparator shifts its result down to meet the bits coming into the next lower comparator. The active and idle comparators alternate vertically as well as horizontally, so that on each beat the active comparators form a checkerboard pattern as shown in Figure 5.

Algorithm design—cell algorithms. Two kinds of cells must be designed to build a pattern matching chip exhibiting the data flow described above:

- The *one-bit comparator* has one bit of the pattern flowing from left to right, one bit of the string flowing from right to left, and the comparison result for the pair of characters flowing from top to bottom. The cell uses this algorithm to update the comparison result:



- The *accumulator* receives d_{in} (the result from the comparator above), λ_{in} (the end-of-pattern indicator), and x_{in} (the don't care bit). It maintains a temporary result t, and at the end of the pattern uses t to replace the result r that flows from right to left:



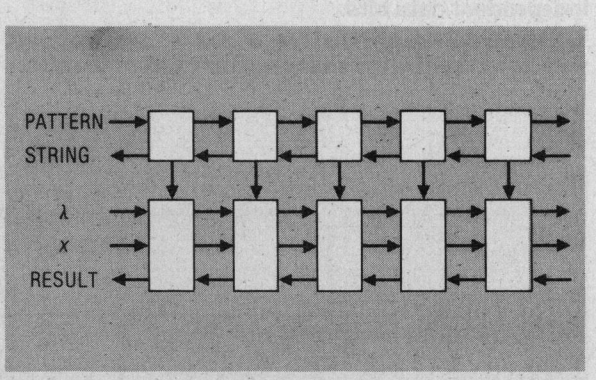


Figure 4. Pattern matching achieved in two modules, each consisting of a linear array of identical cells. Comparators are on the top and accumulators on the bottom.

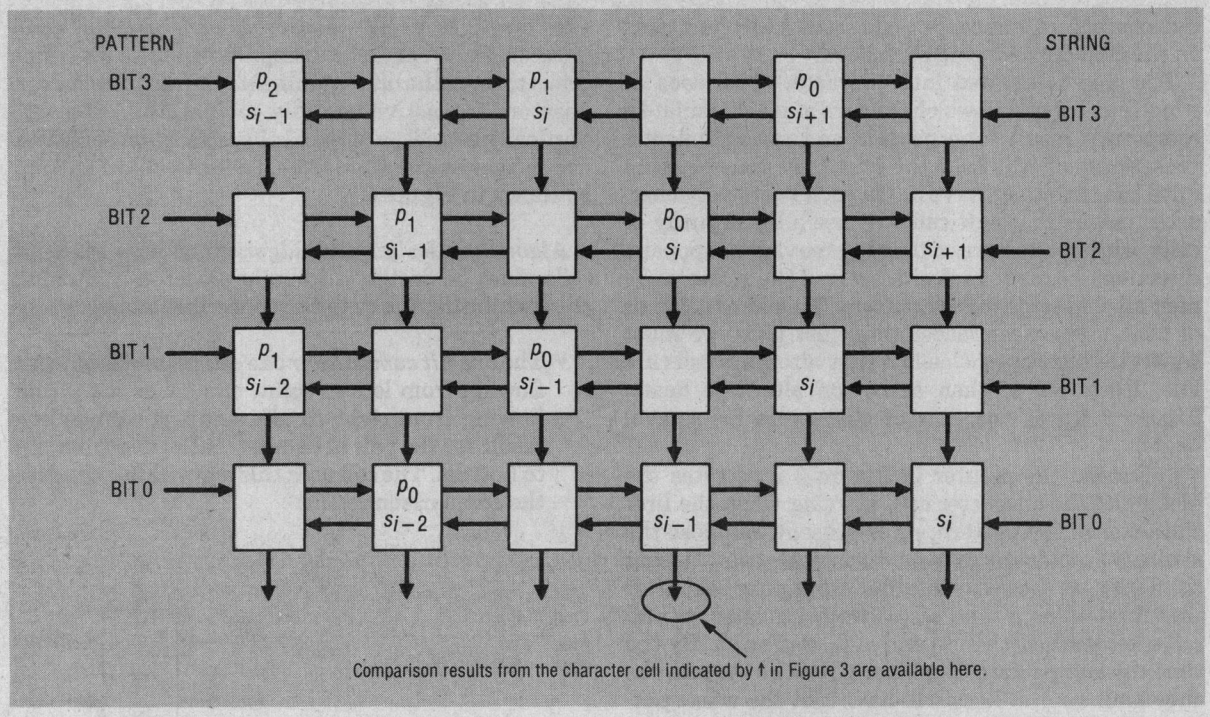


Figure 5. Comparators for single bits.

Circuit and layout design—data flow circuit. Each pipeline used by the algorithm for data flow is implemented as a unidirectional shift register shifting on each beat. Every other cell of the shift register contains valid data. In the NMOS technology used for this chip, a shift register is a chain of inverters separated by pass transistors as shown in Figure 6. When the voltage on the gate of a transistor is near the supply voltage V_{dd}, its channel conducts current, while if the voltage is near ground it does not. The inputs to the inverters can store charges, so data is stored within the inverters; the pass transistors control the inverter inputs. A clock with two non-overlapping phases controls the pass transistors. Because adjacent transistors are turned on by opposite phases of the clock, there is never a closed path between inverters that are separated by two transistors. Alternate inverters can therefore store independent data bits.

The dynamic alternation of active and idle inverters in the NMOS shift register mirrors the alternation of active and idle cells in the algorithm (compare with Figure 5). Each cell can thus contain one gated inverter from each of the shift registers that passes through it. The clock controlling the shift register stages in a cell can activate the cell. The shift register components are then fully utilized—all idle inverters are in idle stages.

Circuit and layout design—cell circuit. Since each cell inverts its inputs before sending them to its neighbors, two versions of each cell must be constructed. One version operates on positive inputs to produce inverted outputs, while the other computes positive outputs from inverted inputs. Transforming a cell algorithm to its inverted twin is straightforward, so the existence of two versions presents no problem. Using the cell algorithms, we can design circuits for the twin versions of each cell. From the circuit designs, we can lay out the masks for fabricating the chip. The positive version of the comparator cell illustrates the process. It takes positive inputs and

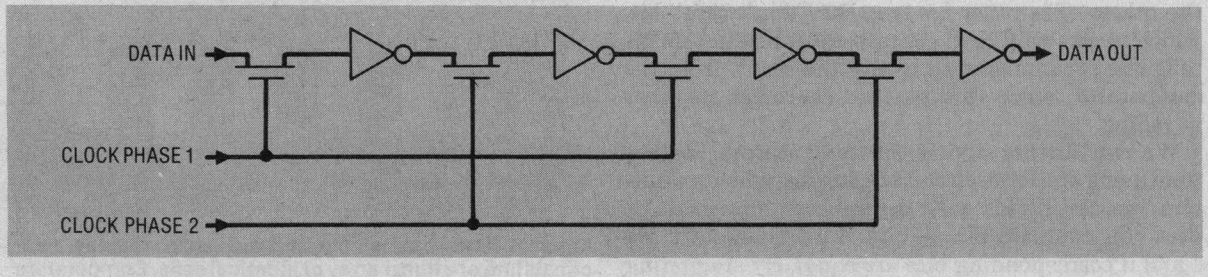


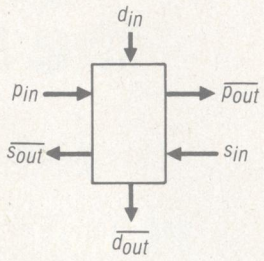
Figure 6. A shift register in NMOS.

produces inverted outputs, so the outputs in the comparator algorithm must be inverted:

$\overline{p_{out}} \leftarrow \text{NOT } p_{in}$

$\overline{s_{out}} \leftarrow \text{NOT } s_{in}$

$\overline{d_{out}} \leftarrow d_{in} \text{ NAND } (p_{in} = s_{in})$



In NMOS, data storage can take place on the input to any logic gate, as long as a pass transistor can isolate that input. The *p* and *s* shift registers can be implemented with inverters as planned, but a NAND gate can be used as the stage for the *d* shift register. Figure 7 is the circuit for the positive comparator. When the clock input goes from ground to V_{dd}, the power supply voltage, all three pass transistors turn on. The pattern and string inputs are then stored on the inverters, and the *d* input is stored on one input to the NAND gate. The exclusive NOR gate outputs TRUE if the two inputs are equal, and FALSE otherwise. The output of this equality test goes to the other input of the NAND gate, which computes d_{out}. After the inputs have stabilized, the clock goes to ground. The outputs of this cell then provide stable inputs to neighboring cells until the clock goes high again.

Circuit and layout design—cell sticks. The next step after completing the circuit diagram is the design of the cell's topological layout, or stick diagram, which shows the relative positions of all signal paths, power connections, and components but hides their absolute sizes and positions. Most of the circuit's components can be implemented in several ways, and a choice among these must be made at this stage of the design. Figure 8 is an example of a stick diagram.

Silicon-gate NMOS technology uses three conduction layers (differentiated by color in Figure 8). Following Mead and Conway's [1] convention, blue lines represent metal conduction paths, red lines polycrystalline silicon (polysilicon), and green lines diffusion into the substrate. The three layers are insulated from each other except at contact cuts, represented by round black dots. The yellow squares are areas of ion implantation, used to create depletion mode transistors. These serve as pull-up resistors in the gates and inverters.

NMOS field-effect transistors are created by crossing a diffusion path (green) with a polysilicon area (red). The green path is the channel, and the red area is the gate. If no ion implantation is present, the channel conducts current only when the gate is at V_{dd}.

The positive comparator cell uses pass transistors and inverters to implement the shift registers; it also uses a NAND gate and an equality, or NXOR, gate. These basic components are combined as shown in Figure 8 to produce the stick diagram for the positive comparator cell. Power and ground run horizontally across the cell on metal (blue) paths. The clock is in polysilicon (red) at the top and right edges, and dips below the upper power wire near the middle of the cell

to allow the cell above to connect to the power wire. Data paths for p and s run horizontally along the top, while d runs downward in diffusion (green).

Let us trace the p data path through the cell. It enters at the left in diffusion and passes through the channel of a transistor that is gated by the clock. Contact is made to a polysilicon path that goes to the input of the p inverter. The inverter output, in metal, crosses the d data path with no interaction and provides an input to the equality gate. It then passes over the s inverter and leaves the cell at the right.

Circuit and layout design—final layout and masking. When stick diagrams have been designed for all of the cells, actual layouts can be produced. These follow the topology of the stick diagrams, but also include the absolute sizes and positions of all components. Designing a layout involves choosing electrical parameters for all transistors as well as following minimum spacing rules for the intended fabrication process. Care must be taken to line up power connections and data paths that cross several

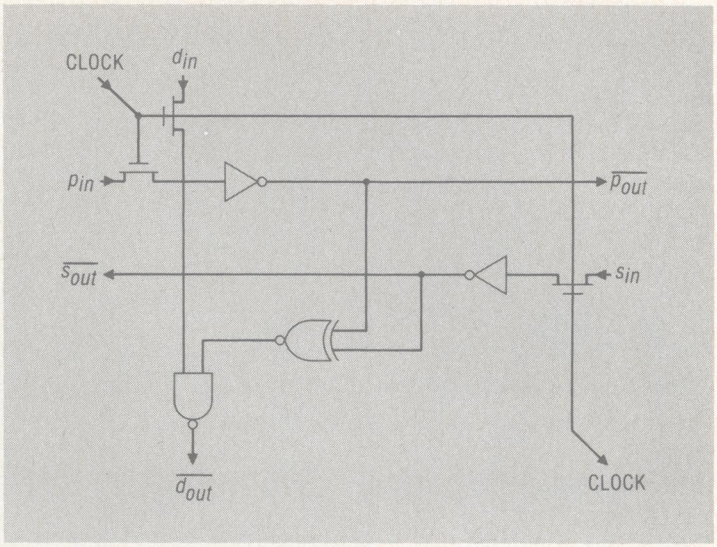


Figure 7. Positive comparator circuit.

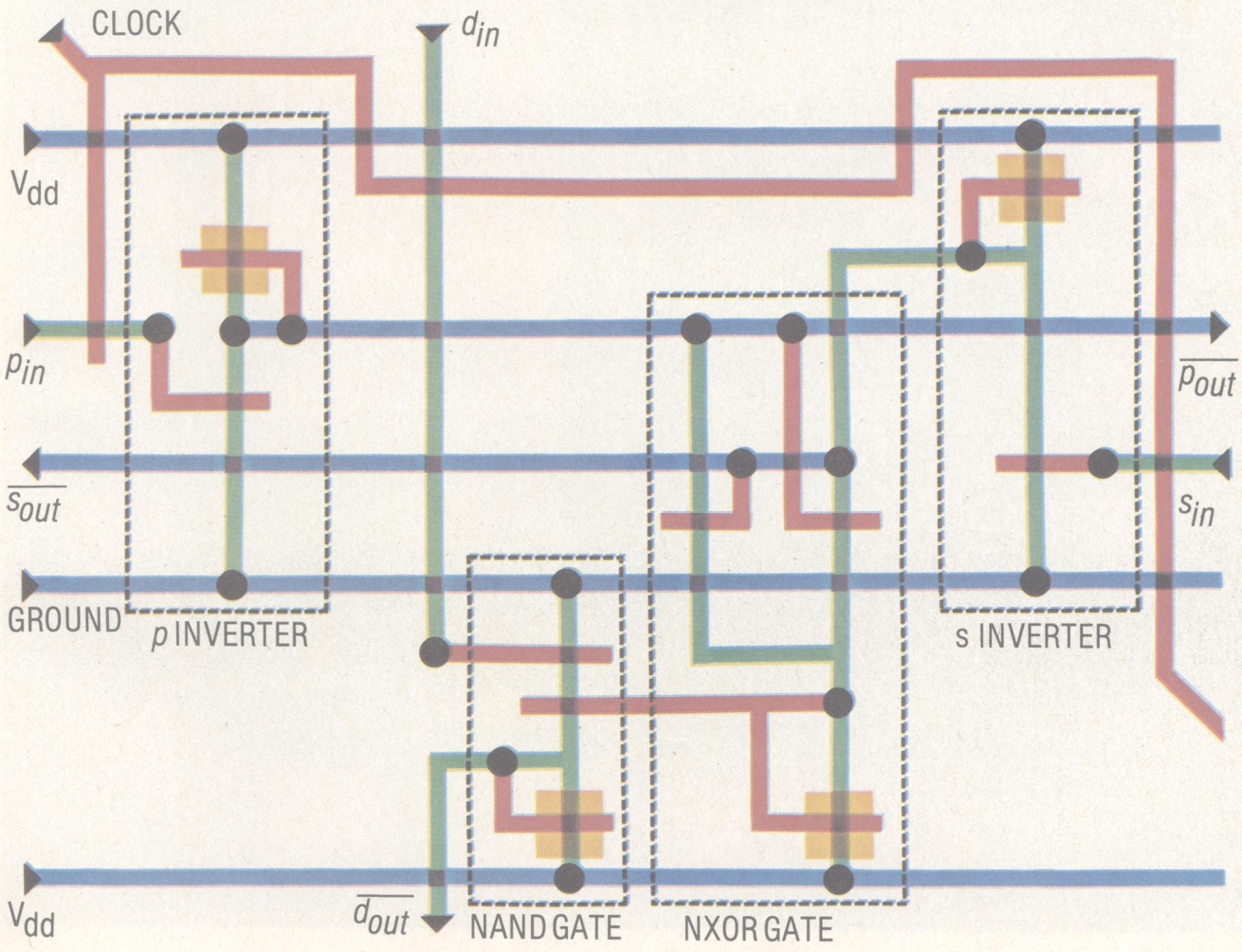


Figure 8. Stick diagram for the positive comparator cell in the pattern matching chip. Color differentiates conduction layers.

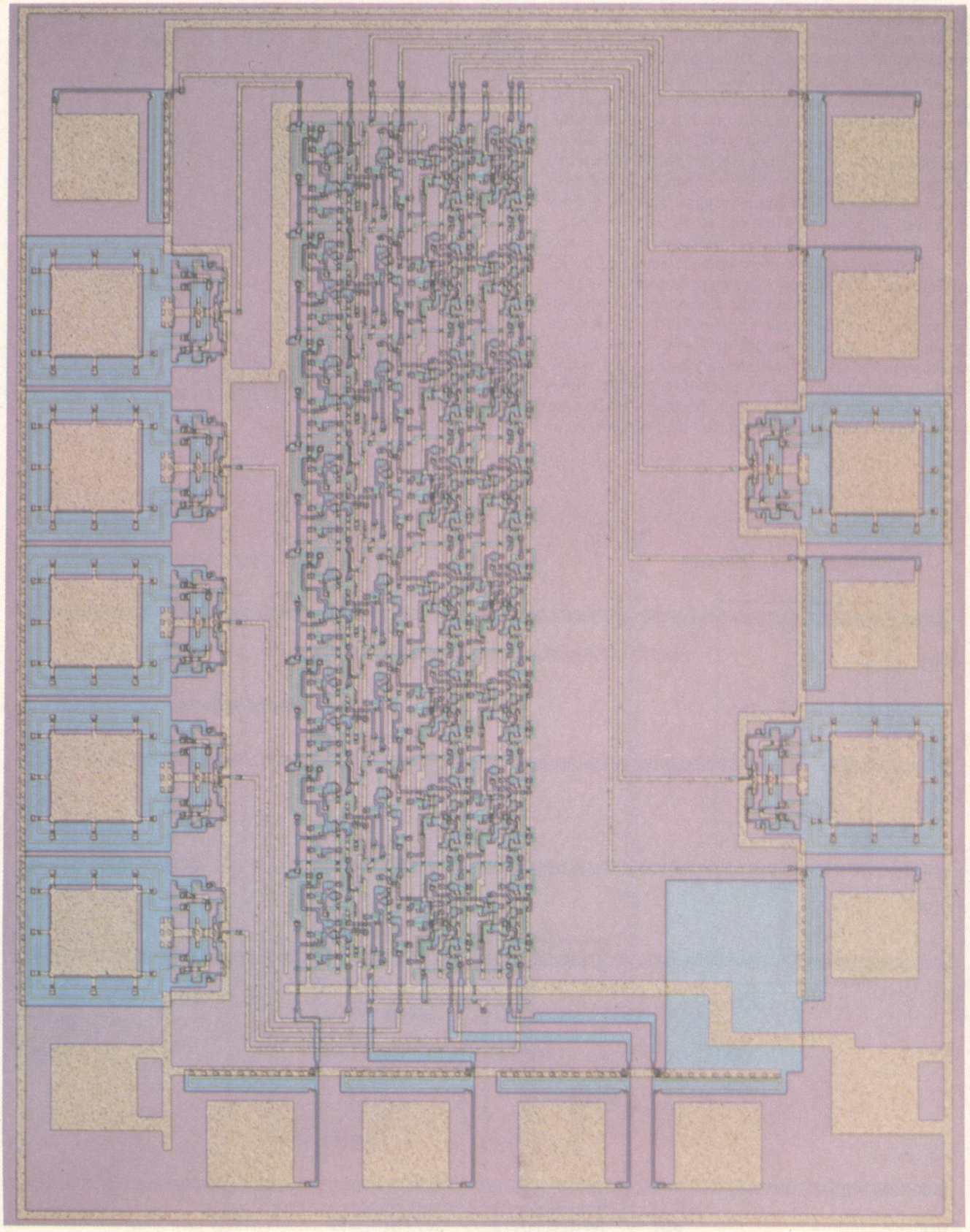


Figure 9. The CMU pattern matching chip prototype. The pattern matching array at left center measures 472 by 1528 microns and is connected to bonding pads on a rectangle measuring 1536 by 1884 microns.

cells. In principle, the layout can be designed mechanically from the circuit and stick diagrams.

When the layouts for all cells are complete, they can be assembled into a working array with the inputs and outputs hooked to contact pads. The layouts can be described using a graphics language (such as the Caltech Intermediate Form[1]) that can be interpreted to make the masks. These masks can then be used to fabricate the chips.

Figure 9 is a photograph of the prototype pattern matcher we constructed according to the methodology outlined above. It can handle patterns containing up to eight two-bit characters.

Design alternatives. In designing the chip we often reached points where we had to choose among several alternatives. There were three major decision-making areas—choice of an algorithm, choice of a data flow implementation, and choice of a method for cell implementation.

Alternative algorithms. A bewildering variety of algorithms could form the basis for a pattern matching chip. The desire for simple and regular data flow rules out the fast sequential algorithms described by Boyer and Moore[9] and Knuth et al.[10] Since these algorithms require dynamically changing communication, their hardware implementation will be too complex to be modular.

Mukhopadhyay[11] has proposed several machines which store a character of the pattern in each cell and which broadcast the text string character by character to all cells. This broadcasting is the major disadvantage of this algorithm. Each cell requires a connection to the broadcast channel, increasing the power requirements of the system as a whole or decreasing its speed. Our algorithm requires no broadcasting of data.

A chip designed by Mead et al.[12] uses another algorithm in which pattern characters are stored in the cells. The text string passes through all of the cells, and the results of character matches are combined using a common wired-NOR bus. We wished to avoid unbounded fan-in of this type, since it may degrade performance when a design is extended to VLSI.

Another algorithm—similar to ours—uses a linear array of cells with data flowing in only one direction. The pattern is permanently stored in the array of cells, and the text string moves past it. Partial results move at half the speed of the text so that they accumulate results from an entire substring match. We rejected this algorithm because of the static storage of the pattern—loading the cells in preparation for a pattern match would require extra time and circuitry.

Our algorithm is well suited to VLSI implementation. All communication is local, since each character cell communicates only with its left and right neighbors. This enhances modularity and extensibility, as well as avoiding the large drivers needed for long-range transmission. Only a few types of cells are used, with many copies of each type. By replicating the basic cells, pattern matching chips of any size can be formed. Finally, control of the chip is simplified since our algorithm requires no separate operation to set up the system for a new pattern.

Alternative data flow implementations. Although the global flow of data is determined by the choice of algorithm, several methods of implementing the data flow may be possible. Serial or parallel data transmission between cells may be selected, for example. Communication may be coordinated in several ways. The data flow can even be transformed to combine several cells into one circuit. We will discuss two of the choices that arose in implementing the data flow of the pattern matching chip.

The existence of idle cells can be avoided by combining pairs of neighboring cells when implementing the data flow. Because each cell pair contains one active and one idle cell at each beat, the two cells can share circuitry. In the pattern matching chip, for example, neighboring comparators could have shared the equality gate, and the d data path could have been multiplexed.

If the amount of sharable circuitry is large enough, it may be advantageous to combine two or more cells in this way. Some additional circuitry will of course be needed to coordinate the sharing and may wipe out the savings. The increased interdependence of the circuit components may also offset the savings, since design changes may become more difficult and errors may be made. The pattern matcher cells are too small to profit from this data flow transformation.

Another choice in data flow implementation is between self-timed and clocked (synchronous) data paths. In a clocked data flow implementation, all data movement is under centralized control. The data flow controller sends signals to each cell to enable data transfers. The pattern matching chip uses this method. In fact, the data flow control signals are the same clock signals needed for data refreshing, although this need not be true in general.

In a self-timed implementation, data flow control is distributed among the cells so that each cell controls its own data transfers. Neighboring cells must obey a signaling convention to coordinate their communication. Self-timed data flow has advantages in modularity and extensibility, since no common clock is needed. Each of the cells may run at its own pace, synchronizing with its neighbors only when communication is needed. Self-timing's disadvantage lies in the extra circuitry needed to implement the signaling conventions. For systems small enough to use a common clock—like the pattern matching chip—clocked data flow is best. For larger systems, of course, self-timed communication may be the better choice.[13]

Alternative cell implementations. Two major choices affected the design of the cells. We rejected static shift registers, which can hold data for long periods without shifting it, in favor of dynamic shift registers, which can not. Also, we chose a random logic implementation of the cell circuitry rather than a more structured approach using standard PLA—programmed logic array—and register layouts.

The dynamic shift registers we used can not hold data for more than about 1 millisecond without shifting. Data is refreshed only by shifting. Static shift

registers, the alternative choice, have regeneration circuitry in every stage so that data can be held indefinitely without shifting. In addition to the two clock phases, static registers need a third signal for the shift command.

Static shift registers are probably the better choice for most systems. They do not invert data between stages, as do dynamic shift registers, and they simplify testing. For our chip, however, dynamic shift registers have advantages. The alternation of active and idle cells allows just one inverter from each shift register to be placed in each cell. This permits the two-phase clock to do double duty as a data flow control signal. The cells and the global layout are thus greatly simplified.

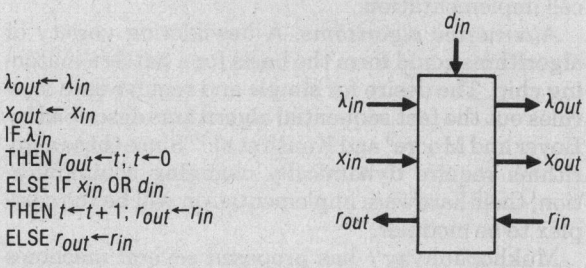
The simplicity of the cell functions dictated the use of random logic. If cells contain more than a few gates, the state-machine design approach should be taken. The state of the cell can be held in a register, and the combinational logic used for changing states can be implemented with a PLA. Standard layouts for registers and PLA cells are available, simplifying design and layout tasks and shortening design change and error correction time. However, the small size of our pattern matcher cells, each containing only four gates, made the use of random logic possible. Design and layout of such simple circuits is easy.

Uses and extensions of the pattern matching chip. A pattern matching chip with n character cells can directly match patterns of length only up to n. Longer patterns require the existence of more than n partial results at each beat. Since any chip must be of finite size, it is important that the chip be extensible. It should be possible to combine several chips to form a larger pattern matcher.

In order to make the chip extensible, an input for the result stream and outputs for the pattern and text streams must be provided. Several pattern matching chips can then be cascaded (Figure 10). The inputs to each chip in the figure are taken from the outputs of its neighbors, so that the cells on all of the chips form a single linear array. The pattern is fed to the inputs of the leftmost chip, and the text string is input to the rightmost chip. The result output is taken from the leftmost chip. A cascade of k chips with n cells each can thus match patterns of up to kn characters.

If the pattern to be matched is longer than the capacity of the available pattern matching system, the pattern can be run through the system several times to match it against the entire string. If the system contains a total of n character cells, each run will match the complete pattern against n substrings. To cover all substrings, all we need do is delay the string by n characters on succeeding runs.

Modifying the design of the pattern matcher can provide special-purpose hardware for problems similar to string matching. For example, we might wish to count how many characters in a substring match corresponding characters in a pattern. This problem can be solved by replacing the result bit stream with a stream of integers, and replacing the accumulator cell with a counting cell:



A problem of more practical interest is the computation of correlations. Here, the pattern, string, and result are all numbers. The result r_i of a correlation is defined as

$$r_i=(s_{i-k}-p_0)^2 + (s_{i+1-k}-p_1)^2 + \ldots +(s_i-p_k)^2.$$

A good match of substring to pattern results in a high correlation.

Correlations can be computed by a machine with a data flow identical to the string matching chip, except that all streams contain numbers. The comparator is replaced by a difference cell that computes

$$d_{out} \leftarrow s_{in}-p_{in}.$$

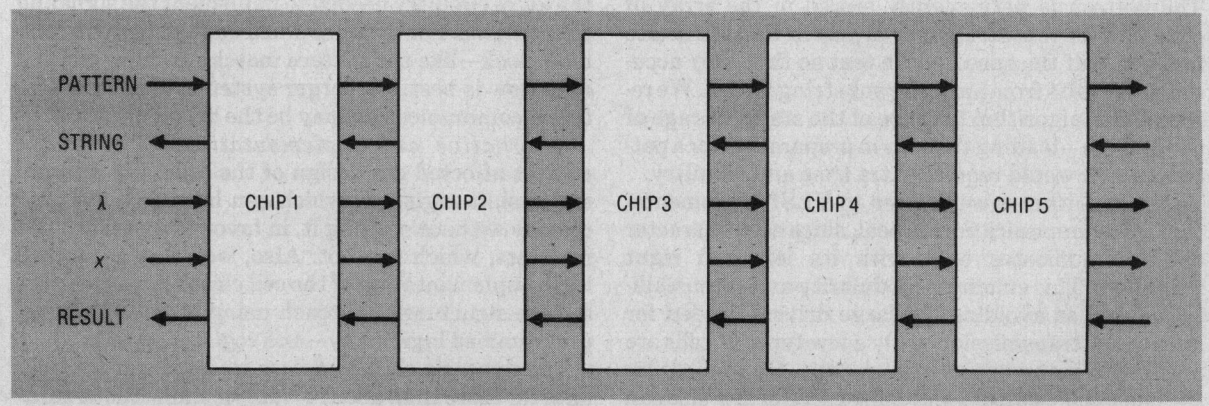


Figure 10. A five-chip pattern matcher—cascading of chips permits the direct matching of longer strings.

Like the character comparison, this difference computation may be pipelined bit by bit. An adder cell replaces the accumulator. The algorithm for the adder cell is

$$\begin{aligned}&\text{IF } \lambda_{in}\\&\text{THEN } r_{out} \leftarrow t;\ t \leftarrow 0\\&\text{ELSE } r_{out} \leftarrow r_{in};\ t \leftarrow t + d_{in}^2.\end{aligned}$$

Other problems such as convolutions and FIR filtering have algorithms using the same data flow.[4,5] It should be clear that special-purpose hardware similar to the pattern matching chip can be built for any of these problems.

Design methodology

A systematic approach is essential when designing a complex system of any kind. The design task must be broken into manageable subtasks, with a well-defined flow of information between them. Each subtask can then be performed separately with no need to consider more than one subtask at a time. This allows division of labor and, more importantly, prevents mistakes and eases design changes.

Because of the diversity of tasks and concerns in VLSI design, a systematic method is especially important in designing a special-purpose chip. It is impossible, for example, to take global data flow, circuit design, and transistor characteristics into account all at once. We must find small subtasks, with boundaries between them that hide the implementation details of one from another. Of course, any set of subtasks is unlikely to be completely independent, since problems that crop up in performing one may require redoing another—difficulties in layout, for example, may mandate a circuit redesign. However, these design iterations will be easier if the interactions between subtasks are few.

VLSI system structure suggests several natural information boundaries. One advantage of geometrically regular algorithms is the spatial separation that they impose between subsystems. The interior of one cell can be designed in ignorance of the interior details of another (although exterior details such as size and data path positions must be known). If cells are complex, the separation of circuit functions within each cell may provide an additional information boundary. The design of each functional block of a cell can then be largely independent of the others. The existence of a hierarchy of abstract chip models, from algorithm to gate to layout level, is a further aid to VLSI design. Each level of the hierarchy deals with an independent set of design issues and serves as an implementation of the next level up and as a specification for the next level down.

Chip design can thus be decomposed geometrically, functionally, and hierarchically. These decompositions must be consistent to be used to best advantage. Tasks separated geometrically should also be separated functionally and hierarchically. It would be unfortunate, for example, if all cell circuits had to be considered at once in order to construct a stick diagram for a chip. Careful construction of a *task dependency graph,* before beginning the design, avoids this problem. This graph should contain all of the subtasks to be performed and include the information needed for each and the precedence relations among them. Of course, backtrack paths resulting in several iterations of one task because of difficulties in another need not be shown. The chip design task is not yet understood well enough to predict such backtracking.

The task dependency graph ensures that no more than a small amount of knowledge is required for any subtask. Each of the subtasks in the graph should deal with the design of one geometric area at one level of abstraction. The circuit design of the entire chip all at once is too large a task because it covers too much chip area. Generating a layout from a cell's function is too large a task, since it spans too many levels of the hierarchy. Designing a single cell circuit from a cell's function is probably a task of the proper size, although if the cell performs several different functions the task should be further subdivided.

Figure 11 is a task dependency graph for the design of a pattern matching chip like ours. Our own project in fact brought out the need for the task dependency graph and suggested its structure. It should be suitable for designing other chips of about the same scale. Each subtask deals with only one geometric region, one circuit function, and one level of the VLSI chip model hierarchy. The arrows indicate the flow of information between the subtasks, each of which we briefly describe below.

Data flow and cell type function. The chip design must begin with an algorithm design conceptually specifying the overall chip structure. Several algorithms will exist for any problem, and the best one should be found at this stage. The algorithm is a level of abstraction at which to think about important properties such as regularity and modularity, without worrying about low-level issues. It should integrate two distinct bodies of information. One is the data flow pattern, including the number of cells, their geometric placements, and the choreography of data. The types of cells should be distinguished and the beats on which each is active should be identified. The other body of information is the function of each cell type, comprising not just the circuit function but also the relative positions of signal inputs and outputs and the sequence of activity on each beat.

Cell combinations and placements. Cells in the implementation might not correspond one to one with the cells in the algorithm. Several cells may be combined to share components or rarely used communication paths in the algorithm may be multiplexed onto one physical data path. The first task in implementing the algorithm is to choose among these combinations and to position cells and cell combinations on the chip.

This subtask requires information about the pattern of active and idle cells on each beat and the use of

each communication path, which the data flow and geometry subtask provides. It also requires information about the sharable subfunctions and complexity of each cell type, which the cell function subtask provides. The output of this subtask is a skeleton layout for the chip, with each cell group assigned a location and a set of contained cells.

Data flow control circuit. Data flow control circuitry ensures the orderly movement of data on the correct beats. To perform this subtask we must learn the correct sequence of beats from the algorithm data flow and which elements are active on each beat. From the cell combination subtask we learn which cell groups and physical data paths contain the active elements.

Based on the size and intended use of the chip we can decide whether the data flow should be clocked or self-timed. If we choose to clock we must decide whether to generate the signal externally or on the chip. We can then design the shift registers for data movement and route any clock wires or synchronization signals among the cell groups on the skeleton layout.

Cell logic circuits. We now have the three pieces of information needed to design circuits for the cells—the cell functions (from the cell type function step), the group of cells to be implemented by each circuit (from the cell combination step), and the shift register stages that must be included in each cell (from the data flow control). If the cell functions are simple enough, ad hoc circuit design techniques may be adequate. If the functions are complex, the cells may be split into subsystems to be designed independently. In this way, full advantage may be taken of the functional decomposition of each cell. In addition, the circuit for each cell type can be designed without reference to the others, since all communication needs have been considered in the data flow control. In designing the circuits, however, consideration must also be given to how the chip will be tested after fabrication.

Cell timing signals. A cell function may comprise several distinct sequential steps performed on each beat. In the pattern matching accumulator, for example, the assignments

$$r_{out} \leftarrow t;\ t \leftarrow \text{TRUE}$$

must take place in the correct order. The cell circuit, especially in a clocked system, may require signals to

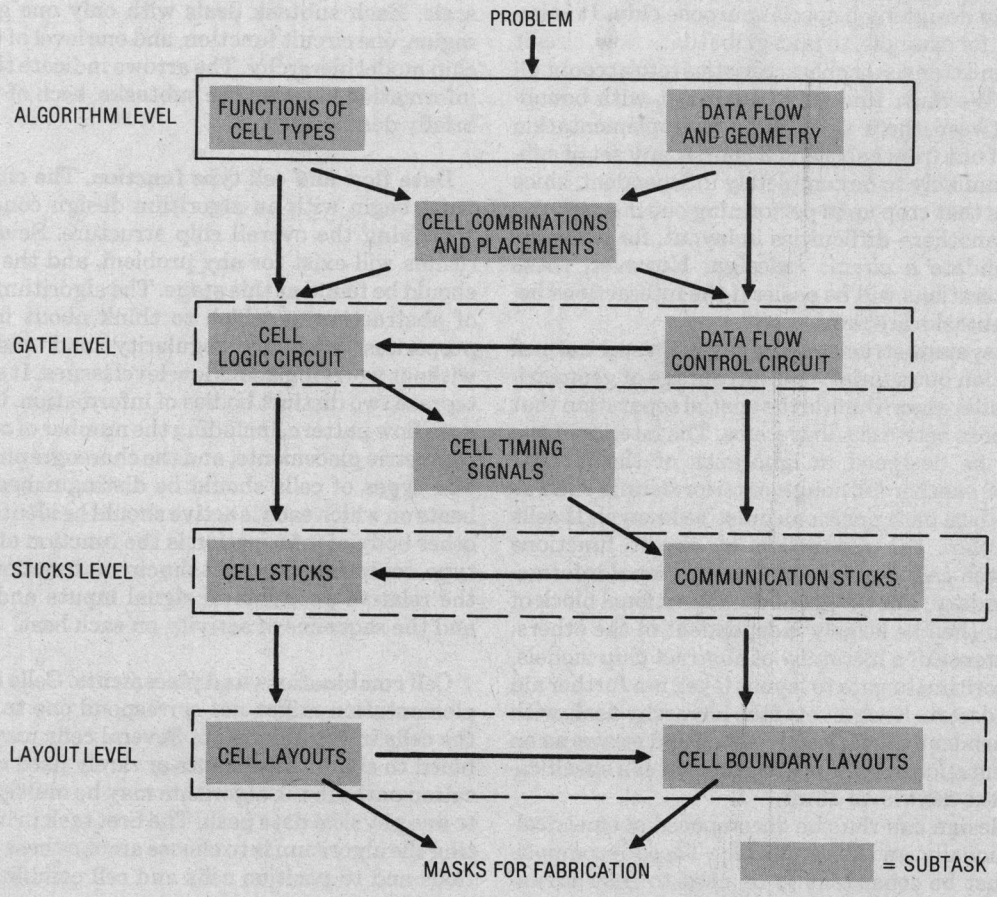


Figure 11. A task dependency graph can guide the designer of a special-purpose VLSI chip. This graph—for a pattern matching chip—shows how the design effort can be decomposed into simpler tasks.

control such sequences, in addition to the signals needed for cell activation. These should be supplied by the data flow control. Such signals should be identified as soon as the cell circuits are complete, and circuits to generate them added to the data flow control.

Communication sticks. When the data flow control circuitry is complete we can draw its stick diagram. For geometrically regular chips this will consist of an open communication path network, control circuitry, and blank spaces for the cells. If the chip has centralized clock circuitry, its topology can be designed. The distribution network for power and ground should also be designed at this stage.

Cell sticks. The topological layouts of the individual cells can now be designed. The relative locations of power, ground, and all inputs and outputs are known from the communication sticks. We must now choose implementations for the circuit elements and decide on the relative positions of internal data paths.

Cell layouts. Once the topological layouts of the cells are complete, the detailed layout of each cell is possible. Following the design rules for the intended fabrication process, actual dimensions for each electrical component and distances between circuit elements must be chosen. This subtask's output is a scale drawing of the cell.

Cell boundary layouts. With cell sizes known, the cell boundaries can be laid out. The communication path and data flow control topology is known from the communication sticks. Wire lengths and spacings can be chosen, as can distances between cells. Inputs and outputs can be connected to contact pads. The cell and cell boundary layouts form a complete description of the chip; once they are complete, masks can be made and the chip fabricated.

Summary of the design methodology. With the help of the task dependency graph, the seemingly complicated process of designing a special-purpose chip can be carried out systematically, one subtask at a time. The graph presented here, although based on limited design experience, seems to be a good starting point. The design tasks below the algorithm level are relatively routine and may at least in principle be helped a great deal by various (future) computer-aided design systems. Eventually only the algorithm design level will require substantial effort and experience.

Over the past few years efforts in several fields of computer science have converged to make possible the design of special-purpose chips as described in this article. The study of parallel algorithms, particularly those for mesh-connected computers, has provided techniques for VLSI algorithm design.[14] The work of Mead and Conway[1] in developing structured NMOS design techniques has eased the design of reliable circuits and layouts. Improvements in computer-aided design and graphics systems have reduced the drudgery of mask design. Finally, the development of suitable intermediate languages makes the design and fabrication processes relatively independent and allows several users to share designs.

These developments allow the relatively inexperienced designer to develop chips quickly and confidently for his own application. By concentrating on algorithms, he can construct—with minimal design time—chips of good performance and fairly small area. The design of the pattern matching chip described here took only about two man-months.

Further developments can make the designer's task even easier. It is possible, for example, to build libraries of standard cells similar to subroutine libraries. If a designer needs, say, an inner product step cell, he can select it from a library rather than construct it himself. Libraries of data flow implementations are also possible, although their forms are less obvious.

Advances in fabrication technology may increase the scale of projects that can be attempted. Aside from reductions in feature size, techniques such as wafer-scale integration will increase the size and power of special-purpose devices. The modularity inherent in our philosophy is especially appropriate to wafer-scale integration, where a wafer's chips are in-

terconnected rather than cut apart for individual packaging. Since some of the wafer's chips may be defective, the fabricator must be able to reroute the interconnections to replace a faulty chip with a functioning one. He can do this easily if the chips have regular interconnections and if they include only a few types.

The philosophy and methodology described here will make practical the design of special-purpose VLSI chips by their users. Connected to a general-purpose computer, these devices will provide rapid solutions to a variety of computations. Given the ease of the design process and the availability of new design tools, VLSI modularity will become a common architectural strategy in the 80's. ■

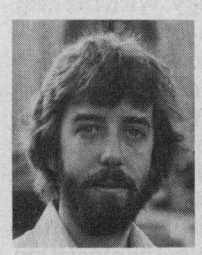
Acknowledgments

We received help from many people during our research. Using a preliminary version of Mead and Conway's *Introduction to VLSI Systems,* Bob Sproull taught us basic NMOS design techniques in his VLSI design course at CMU. Our pattern matching chip prototypes are included in the XEROX PARC multiproject chips for spring 1979 and have been fabricated at XEROX facilities. Larry Stewart checked our layouts for violations of design rules. Bob Hon provided much needed help, including building the layout design system we used, converting our designs into ICARUS format, and mounting chips for testing. Philip Lehman and Siang Song offered suggestions in the early stages of the algorithm design. Lynn Conway, Bob Hon, Dick Lyon, Siang Song, and Bob Sproull reviewed this paper. To these people, and to the others who helped us directly or indirectly, we express our thanks.

This research was supported in part by the Defense Advanced Research Projects Agency under Contract F33615-78-C-1551 (monitored by the Air Force Office of Scientific Research), the National Science Foundation under Grant MCS 78-236-76, the Office of Naval Research under Contract N00014-76-C-0370, and an NSF Fellowship.

References

1. C. A. Mead and L. A. Conway, *Introduction to VLSI Systems,* Addison-Wesley, Reading, Mass., 1980.
2. R. Hon and C. Sequin, "A Guide to LSI Implementation," 2nd ed., XEROX Palo Alto Research Center Technical Report, 1979.
3. I. E. Sutherland and C. A. Mead, "Microelectronics and Computer Science," *Scientific American,* Vol. 237, No. 3, Sept. 1977, pp. 210-228.
4. H. T. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI)," in I. S. Duff and G. W. Stewart, eds., *Sparse Matrix Proc. 1978,* Society for Industrial and Applied Mathematics, Philadelphia, Pa., 1979, pp. 256-282. A slightly different version appears in C. A. Mead and L. A. Conway, *Introduction to VLSI Systems,* Addison-Wesley, Reading, Mass., 1980, sec. 8.3.
5. H. T. Kung, "Let's Design Algorithms for VLSI Systems," *Proc. Caltech Conf. Very Large Scale Integration,* California Institute of Technology, Pasadena, Calif., Jan. 1979., pp. 65-90. Also available as a Carnegie-Mellon University Computer Science Department Technical Report, 1979.
6. R. E. Griswold, J. F. Poage, and I. P. Polansky, *The SNOBOL4 Programming Language,* Prentice-Hall, Englewood Cliffs, N. J., 1968.
7. P. J. Warter and D. W. Mules, "A Proposal for an Electronic File Cabinet," *MICRO-DELCON,* IEEE Computer Society, Long Beach, Calif., Mar. 1979, pp. 56-63.
8. M. J. Fischer and M. S. Paterson, "String Matching and Other Products," Massachussetts Institute of Technology, Project MAC, Technical Report 41, 1974.
9. R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," *Comm. ACM,* Vol. 20, No. 10, Oct. 1977, p. 762.
10. D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast Pattern Matching in Strings," *SIAM J. Computing,* Vol. 6, No. 2, June 1977, pp. 323-350.
11. A. Mukhopadhyay, "Hardware Algorithms for Nonnumeric Computation," *IEEE Trans. Computers,* Vol. C-28, No. 6, June 1979, pp. 384-394.
12. C. A. Mead, R. D. Pashley, L. D. Britton, Y. T. Daimon, and S. F. Sando, "128-Bit Multicomparator," *IEEE J. Solid State Circuits,* Vol. SC-11, No. 5, Oct. 1976, pp. 692-695.
13. C. L. Seitz, "Self-Timed VLSI Systems," *Proc. Caltech Conf. Very Large Scale Integration,* California Institute of Technology, Pasadena, Calif., Jan. 1979, pp. 345-355.
14. H. T. Kung, "The Structure of Parallel Algorithms," in M. C. Yovits, ed., *Advances in Computers, Volume 19,* Academic Press, N.Y., 1980.



Michael J. Foster is working toward a PhD in the Computer Science Department at Carnegie-Mellon University, where he holds a National Science Foundation Graduate Fellowship. From 1973 to 1977 he was employed as a programmer by Tracor Northern and Princeton Gamma-Tech. His current research interests include algorithms, VLSI structures, and computer system architecture. Foster received a BS in mathematics from MIT in 1973 and is a member of ACM and Sigma Xi.

H. T. Kung is an associate professor of computer science at Carnegie-Mellon University, where he has been on the faculty since 1974. His current research interests include special-purpose chip design, VLSI-related complexity theory, and data base systems. He has worked on algorithms for parallel computers and problems in computational complexity. Kung graduated from National Tsing-Hua University, Taiwan, and received his PhD degree from Carnegie-Mellon University.