In spite of its somewhat unlikely sounding name, this complier can effectively design custom ICs from an algorithmic specification of circuit behavior.

MacPitts: An Approach to Silicon Compilation

Jay R. Southard, MIT Lincoln Laboratory

Silicon compilation is a new technique for integrated circuit design that has recently undergone intense development, scrutiny, and criticism. At the moment, there is even disagreement over the meaning of the term "silicon compilation"[1] itself. In this article, "silicon compilation" will refer to the automatic synthesis of an integrated circuit layout from a description of its behavior. Traditional integrated circuit design techniques, on the other hand, are dependent on the *structure*, rather than the *behavior* of the integrated circuit.

Of course, design is not the only step in the production of useful ICs. Fabrication and testing are also important. In fact, advances in fabrication capability have actually outstripped advances in design, which is why design is a topic of such interest at present.[2] Some testing issues—design for testability and automatic generation of test patterns, for example—are potentially part of the design process. Current research is beginning to show progress in these areas, and there is reason to believe that silicon compiler design methodology is more amenable to test automation than more handcrafted methods.

The disparity between fabrication and design is such that while it is economic to fabricate a moderately complex circuit—one that is to be replicated in production about 1000 times—it is not economic to design such a circuit unless it will be replicated at least 10,000 times.* Thus, most IC vendors will not accept commissions for designs without a commitment to a 10,000-unit production volume. Reducing the design cost of an IC by a factor of two would probably reduce the minimum economic production volume to about 1000 units, and a good silicon compiler can probably reduce design cost by much more than that—probably by a factor of three to five (see Figure 1).

*This is a rough calculation based on quotes from silicon foundries for prototype and low-volume production runs of custom integrated circuits compared to similar runs of printed circuit boards.

The design of integrated circuits

There are at least two separable aspects to IC design. One is the rigorous specification of the integrated circuit's function. Another is the layout of the active devices and interconnections required to instantiate the circuit. This layout description assumes some specific or generic IC fabrication technology. Because going directly from behavior to layout is usually too complicated to accomplish in one step, designers customarily construct a circuit description that lies somewhere between the behavioral and layout descriptions of a specific circuit. The amount of engineering time spent in each of these three design areas is shown in Figure 1.

During the past 15-20 years, great strides have been made in the understanding and utilization of program like behavior specifications. It was quickly discovered that the majority of the functions of even the most complex digital hardware system could be described by a modest program. Thus, the main thrust of research activity must be to improve layout and circuit design productivity.

Below, we will connect these factors to the evolution of the MacPitts[4] silicon compiler. For those desiring a still more comprehensive survey of integrated circuit design methods, one can be found in the literature.[5]

Layout

In simple designs, the layout is often obvious to a good designer. In more complex designs, however, there are a number of reasonable layouts to consider. Additionally, the functions of medium-scale integrated circuits can often be circuit designed in several intuitively reasonable ways. In order to improve productivity, two basic approaches to IC layout design have traditionally been used: computer-aided drafting and layout synthesis from circuit design.

Computer-aided drafting. This layout method recognizes the "fine structure" of the layout design task. In order to achieve efficiency vis-a-vis a specific technology, the real design process must be iterative, as shown in Figure 2, rather than follow the straight flow of Figure 1. A computer-aided drafting tool combined with hierarchical design style can thus be valuable for easing the task of reworking a layout. In addition, the layout specification in the computer can be utilized to provide automatic layout and electrical checks and device simulations.

Because the drafting system and the photomask fabrication process deal with the same objects, the designer has, potentially, complete freedom to utilize all the capabilities of any imaginable process. In practice, however, two productivity implications must be taken into account when laying out a circuit. One factor is implied by a hierarchical design style. If some cells already exist—cells that can be incorporated into a new design—this can result in greatly reduced design time and greatly increased confidence that those cells will work. The other productivity factor is a consequence of the fact that the time required to design a large-scale IC often spans one or two modifications of the target IC fabrication process. Successful integrated circuit production will span many such changes. A highly optimized layout for one process will be suboptimal for its successor. It is valuable to be able to automatically modify otherwise satisfactory masks so that they match an improved fabrication process. Although the modified layouts are generally still suboptimal, automatic modification is usually an acceptable engineering compromise.

Layout synthesis. New layout tools and techniques generally emphasize the above-mentioned practices, even at the cost of some further reduction in the absolute efficiency of the available fabrication technology.

The standard cell layout method institutionalizes the hierarchical borrowing practice. This method provides a cell library composed of logic gates and complex cells. It is common practice to include automatic place-and-route routines in the cell library, meaning that the designer only needs to specify the connectivity between cells, not lay out the interconnection.[6] Again, this is a trade-off of efficiency for productivity.

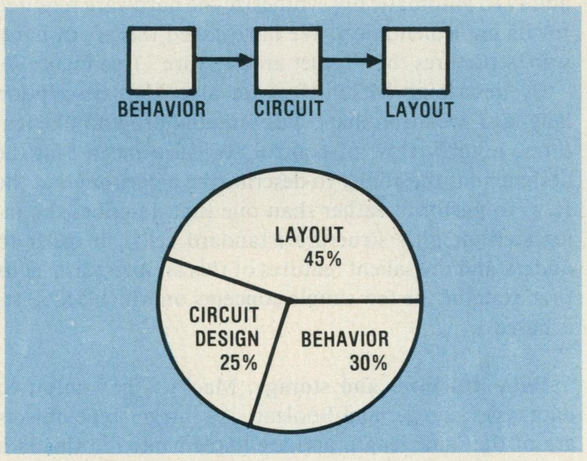


Figure 1. Three levels of IC specification and the time spent in each. The time chart is based on actual measurements.[3]

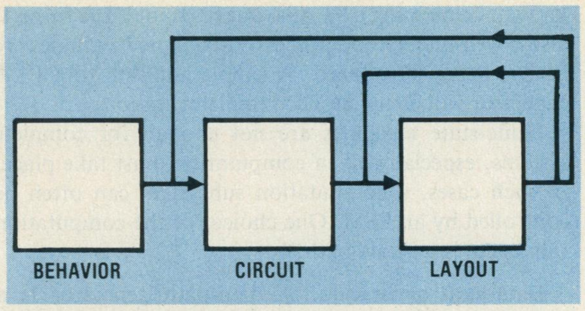


Figure 2. The iterative design process.

Another approach to layout synthesis is sometimes called "symbolic layout." Here, the layout is not specified down to its absolute geometrical location. Rather, relationships are sketched out from which the layout can be automatically generated.[7] The sketchier the relationship specification, the harder the automatic program must work, and the less efficient the resulting layout will be. However, the relationship sketch is a general specification that need not be altered for reasonable changes in fabrication capabilities. Only the layout program will be modified, and that will be used to resynthesize many designs. Although advantageous, these symbolic layout systems still require considerable skill and effort for them to be used properly.

Circuit design: silicon compilation precursors

The difficulty with the layout methods described above is that none of them provide an underlying systemization for transforming behavior into the required structural, circuit terms. In other words, they do not address the problems of circuit design. In this context, two *subsystem* generation techniques deserve special mention. One involves the use of PLAs (programmable logic arrays) in implementing finite-state machines.[8] The other is an offshoot of the standard cell methodology—the "data-path generation" technique pioneered by Johannsen[9] and also recently used by Shrobe.[10]

Subsystem generators must be used in conjunction with other subsystem generators to create a complete integrated circuit. This means that several decoupled descriptions of the same system must be created—one for each subsystem generator. Also, the generated layouts must be combined, either manually or by some "chip assembler" techniques that are not yet generally available.

Finite-state machines and PLAs. Finite-state machines, or FSMs, are a powerful conceptual technique for specifying some forms of control. PLAs are an efficient and easily laid out implementation of logic. A PLA implements a universal form of logic; all logic functions not requiring storage can be implemented in a large enough PLA. Additionally, efficient algorithms for transforming any logic expression into PLA forms are well-known. Generating a PLA layout from the PLA forms is also a well-understood process, though new wrinkles are always being introduced. A simple addition to a PLA generator will create an FSM implementation.

Finite-state machines are not enough for complete systems, especially when computation must take place. In such cases, a computation subsystem can often be controlled by an FSM. One choice for the computation subsystem is a "data path."

Data-path generation. A data-path generator is a specialized standard cell system that systematizes and takes advantage of typical constructions. In a data-path generator, bit-wise subunits are combined to create functional units, and multiple units are combined to create a data path. A data path is a sequence of operators—addition units, multipliers, comparators, etc.—connected to perform a computation. Thus, a systematic transformation is possible from the behavior/computation specification of an IC directly to its structural, standard cell specification. The connection of operators is done by automatic routing of interunit signals.

The systems that use the generated data paths, however, typically require conditional data flows, looping, and other control constructs. The data paths, therefore, must be capable of being switched during operation by signals generated outside the data path. To do this, the data-path generators leave hooks for control signals. A different tool, and, more importantly, an unrelated description, must be used to create these control signals.*

First

Another precursor to general silicon compilation is First.[12] First (fast implementation of real-time signal transforms) was designed for digital signal processing (DSP) applications. First and MacPitts differ in nearly every way—from philosophy of behavioral specification to circuit implementation. Both are alike, however, in that they produce complete chip layouts, including I/O pads and routing.

There are several interesting facets to First, but the one most important is that many DSP applications can be described strictly in terms of data-flow graphs—for example, the nonrecursive filter of Figure 3. Note that there are no conditionals of any kind because *there is no need for them.* This is in contrast to the data-path generators typically used for algorithms and systems that do require conditionals, time-varying interconnections, and so on. First is also superior to most data-path generators in that it can interconnect any network of its functional units. First uses bit-serial operations to implement its functions, and this often yields a superior fabrication cost/function ratio. However, it also results in greater design complexity because of the necessity of taking the bit delay timings into account when connecting functions.

*Recently, Agre[11] reported on work in which a unified high-level description—one similar to MacPitts—was used to generate the descriptions for both an existing data-path generator and a control signal synthesizer.

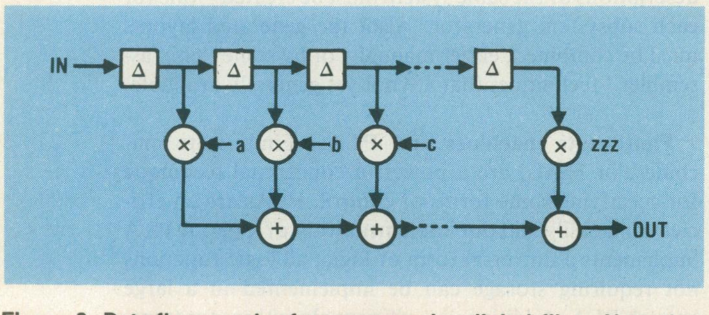


Figure 3. Data-flow graph of a nonrecursive digital filter. Note that there are no options in the flow of data; there is no control flow.

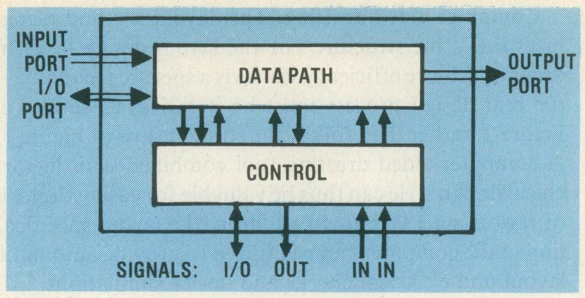


Figure 4. The MacPitts target architecture.

The necessity of considering bit delay timings led the First team into a powerful, formal method of bit-level design. Also, the very restrictive operator combination paradigm allows the First designer to guarantee speed performance. In turn, this guarantee has led to methods for time multiplexing units to match the actual performance with specified performance criteria.

MacPitts

MacPitts was designed to be a synthesizable, algorithm description language. By synthesizable, we mean that the language was designed in conjunction with IC synthesis routines. Therefore, during the evolution of MacPitts, proposed language features and concepts were considered both on the basis of possible application and on our ability to automatically synthesize the hardware required for its implementation. We have called this set of hardware structures the "target architecture" (see Figure 4).

By describing MacPitts as an algorithm description language, we mean that it has essential programlike features, notably flow of control. We also mean that the designer has the ability to describe the algorithm that the IC is to perform, rather than one that describes the integrated circuit's structure (standard cell). In order to understand the salient features of this architecture, let us first examine the few simple concepts on which MacPitts is based.

Data structures and storage. MacPitts has only two data types: integer and Boolean. All integer-type objects are of the same length and are implemented in the data path. Boolean-type objects are implemented in the control and flags sections. Our terminology is based on the following:

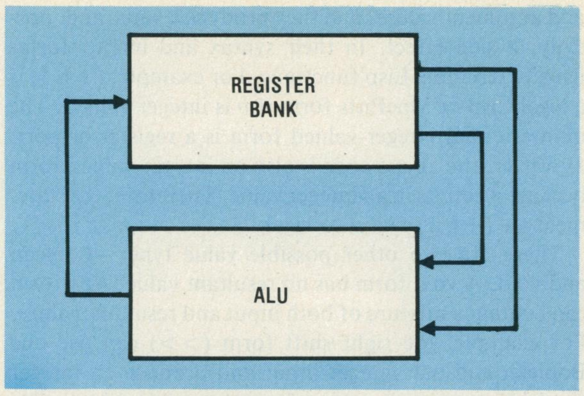


Figure 5. Conventional data-path organization.

Integer	Boolean	
register	flag	stored
port	signal	not stored

Storage is kept in master-slave flip-flops. These flip-flops are called registers and physically exist in the data path if they store integer-type data; they are called flags if they store Boolean-type data. All storage is modified on a single, synchronous "state transition." During a state (between two successive state transitions), the storage devices present a constant output. At the same time, their inputs are allowed to settle. At the state transition, the storage input values are propagated to their outputs. The flip-flops are designed so that the new outputs cannot modify the inputs during the state transition. This is precisely the effect that a programmer expects from the Lisp instruction

(setq a (+ a 1))

or in Fortran

A = A + 1

On the other hand, nonstorage items are designed so that they are modified asynchronously during the state. Thus, the MacPitts instruction

(setq a (+ a 1))

while syntactically valid whether *a* is a register or a port, will only be useful if it is a register.

Data-path operations and transfers. So far, we have only presented the framework in which computation can be accomplished. Because many of the target applications require high throughput and can utilize parallelism, we wanted a design language and a target architecture that would support such parallelism. Our target architecture achieves this parallelism by implementing a high degree of concurrency in data operations and transfers within the data path. For comparison, this concurrency can be considered alongside "standard" and "horizontally microprogrammable" computer architectures.

The conventional computer data path is typically partitioned into a register/memory array and an ALU, as shown in Figure 5. Only one operation can proceed each clock cycle. Efficient algorithm specification in the form of a program, however, is simplified because of this limitation; the ALU merely needs to be kept busy for full efficiency.

A horizontally microprogrammed machine typically has several buses connecting a small number of functional units that can operate in parallel. The microprogram specifies the connection of units to buses on an instruction-by-instruction basis. Although some parallelism is available, there is usually not enough to execute all the parallelism implicit in the algorithm. Both the number of functional units and their allowed interconnection are usually limited. Such restrictions create both programming complications and execution bottlenecks.

For example, in one typical, high-performance microprogrammed array processor, it is not possible to directly sum a series of values from the main data memory because both the main data memory and the adder output are limited to entering the adder at one and the same input port. Efficient programming demands that many of the units (and hence transfer buses) be kept busy simultaneously. Fitting an algorithm into the Procrustean bed of any specific microprogrammable machine is typically an error-prone and difficult programming chore. This chore is complicated by the usual situation of obtaining partial results in a unit that cannot be directly connected to the unit required for the remaining computation.

MacPitts, however, allows a designer to specify an algorithm as though completely general and sufficient parallelism existed in some general-purpose machine. In other words, any control/data-flow graph can be directly specified in MacPitts. Then the MacPitts compiler "extracts" the minimum-hardware microprogrammed machine which executes that parallel algorithm, with all the bus and unit merging and sharing that that implies. The resulting structure is topologically similar to any microprogrammable machine's architecture but is organized as shown in Figure 6. The data-steering control functions are provided in the control section, which is generated from the same algorithm specification by the extraction process. Thus, MacPitts combines ease of programming with the efficiency and parallelism of the horizontally microprogrammed architecture.

From our knowledge of the data-path generators previously discussed, we were aware of the usefulness of constructing the data path out of bit-wise units "glued" together in one direction to form word-length units, which are then bused together in the other direction. A more detailed picture of the layout scheme for the data path is given in Figure 7. A bit-wise unit is called an *organelle.* A standard library of the usual functions—adder, subtractor, shifters, comparators, etc.—is provided. More sophisticated users can design their own organelles. However, as units become more special-purpose, the MacPitts program becomes more a *structural* than a *behavioral* specification.

MacPitts specification language—forms. There are two fundamental concepts to the MacPitts language. The first of these is "state transition," which we have already discussed, and the second is that of a "form." Form includes the syntax and semantics of logic, arithmetic, and control expressions. Forms are composed of an operator

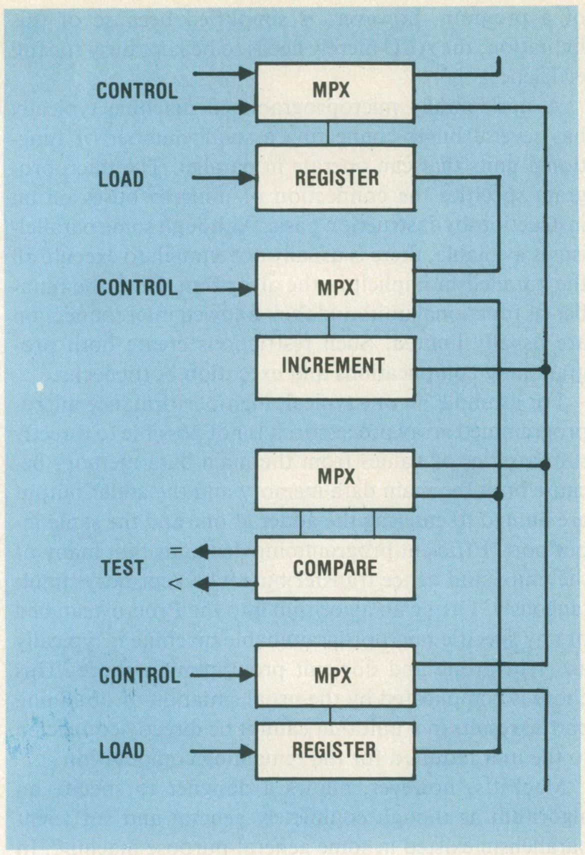


Figure 6. Organization of the MacPitts-generated data path. Note the lines going to and from the control section (not shown). The control section is generated from the same program-like specification.

and argument values, and they produce a value and, possibly, a side effect. In their syntax and intent, forms usually resemble Lisp functions. For example, (+ b 1) is a legal Lisp or MacPitts form if *b* is integer valued. The quintessential integer-valued form is a register or port. However, the above code is also an integer-valued form because it results in an integer value. Therefore, the statement (+ (+ b 1) 1) is also legal.

There are two other possible value types—Boolean and void. A void form has no resultant value. Any form can contain a mixture of both input and resultant values. For example, the right shift form (>>) can use one Boolean and one integer input and produce an integer result. Note, too, that some forms have side effects. The most obvious examples of such forms are assignment statements and control flow branches. These forms accommodate the storage and state transition concepts previously presented. Thus (setq a (+ a 1)) has the desired effect of incrementing *a* after the next state transition.

A compile-time side effect not usually occurring in a standard computer language is that some forms cause other forms embedded in their scope—that is, the forms that make up the arguments of the outer form —to be executed in parallel instead of sequentially. Forms that cannot be executed in parallel can share physical units in the data path. Those that cannot syntactically (at compile time) be determined as mutually exclusive, must be assumed to execute in parallel and thus cannot share physical units. *Parallelizing forms increase the computational throughput of a design at the expense of silicon area.* A later example will demonstrate the use of parallelizing forms in examining the trade-off between throughput and silicon area.

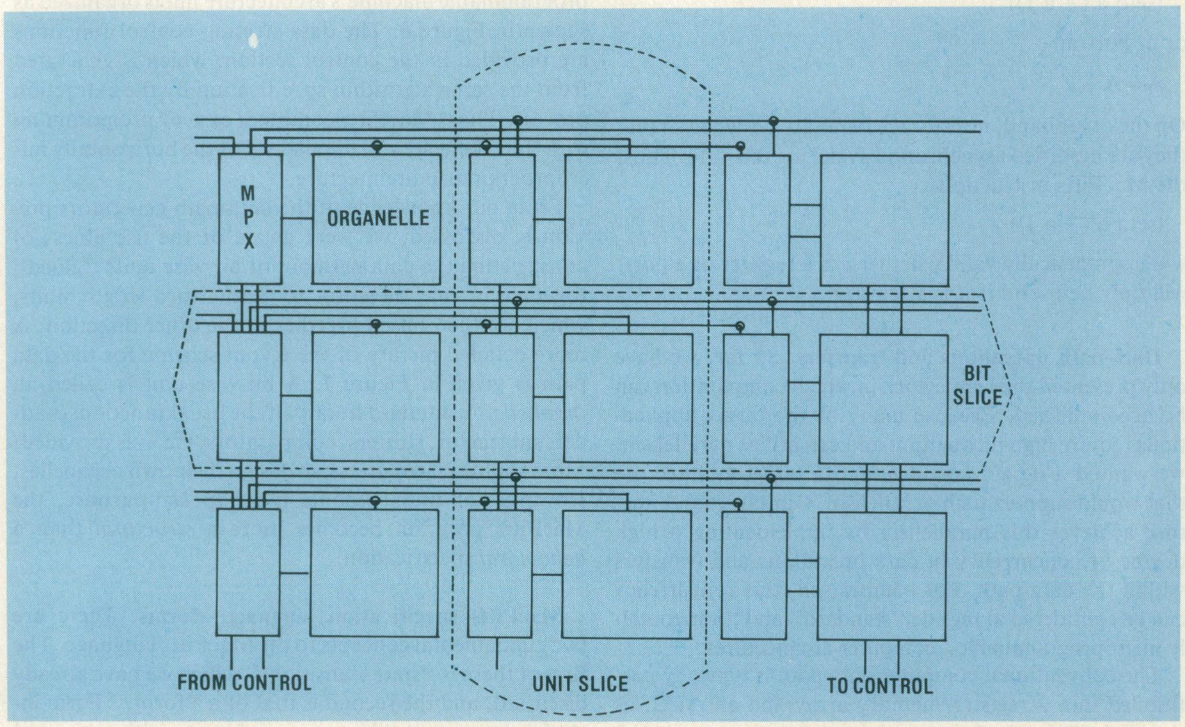


Figure 7. Detailed organization of MacPitts data paths.

The *cond* operation. In MacPitts, the *cond* operation is the method of condition testing for execution and control flow. This operation is probably the most important MacPitts form; it is certainly the most complicated. It is *cond* that distinguishes MacPitts from a data-path generator and makes it an algorithm specification language rather than a data-path specification language. The *cond* operation is also a parallelizing form.

There are several ways of thinking about the *cond* statement. First, it is syntactically identical to the Lisp *cond* statement and may easily be viewed as a program-like "case" statement. This paradigm, however, ignores not only the implicit parallelism of the form, but also a possibly preferable paradigm based on finite state machines.

Consider the following code fragment (assuming that *a* and *b* are registers and *exch* and *incr* are Boolean values):

```
execute
(cond  (exch (setq a b) (setq b a) (go fetch))
       (incr  (setq a ( + a 1))
              (setq b ( + b 1))
              (go fetch))
       (t     (setq a ( − a 1))
              (setq b ( − b 1))
              (go fetch)))
```

MacPitts interprets this specification of the state labeled *execute* as follows: If *exch* is true, exchange the contents of registers *a* and *b* and go to the state labeled *fetch*. (Note that *cond* has parallelized the two *setq*s, otherwise this code would just have set *a* to *b*.) If *exch* is not true, but *incr* is, then increment both *a* and *b* simultaneously and go to *fetch*. Finally, if neither conditional is true, *a* and *b* will both be decremented (simultaneously), and, again, *fetch* will be the next state. Notice that MacPitts has assumed that the programmer wanted all the conditions to be mutually exclusive. This is consistent with the Lisp interpretation of the conditions (predicates), but not the consequent actions. The flowchart equivalent of this code is displayed in Figure 8.

In the *execute* sample, the predicates are simple Boolean values. These would be distributed via the control section, commanding the organelle multiplexers to connect the operators and registers to the correct buses. Instead of Boolean values, any form with a Boolean result could have been used as a predicate:

```
(cond (( = a b) (setq a 0))
      (t        (setq a ( + a 1))))
```

In this case, a comparator in the data-path would compare *a* and *b*. The comparison would result in a Boolean signal that would be distributed via the control section to command other multiplexers in the data path.

The crux of the difference between Lisp and MacPitts interpretations is that Lisp's consequent actions are sequential but MacPitts' are parallel. The entire MacPitts *cond* statement can be, and is, compiled into silicon capable of executing it in a single state cycle. The MacPitts code, and its timing, make eminently good sense in the context of a Mealy-type FSM.* First, let us make the distinction between data storage and storage of FSM state. The *cond* form can then be interpreted as the way MacPitts specifies the next-state and output mappings for the current FSM state. In fact, we can adopt the convention that each FSM state is represented by a set of < condition / actions / transition > triples. Note that, unlike the usual FSM conventions, "actions" may affect data storage as well as specify output. The sample code is thus the equivalent of the triples

$$\begin{aligned} &\text{exch}/a \leftarrow b, b \leftarrow a \;/\; \text{fetch} \\ &\overline{\text{exch}} \cap \text{incr} \;/\; a \leftarrow a+1, b \leftarrow b+1 \;/\; \text{fetch} \\ &\overline{\text{exch} \cup \text{incr}} \;/\; a \leftarrow a-1, b \leftarrow b-1 \;/\; \text{fetch} \end{aligned}$$

for the state *execute*. This is presented graphically in Figure 9.

*The Mealy-type FSM output functions depend on both the current state and the input values.

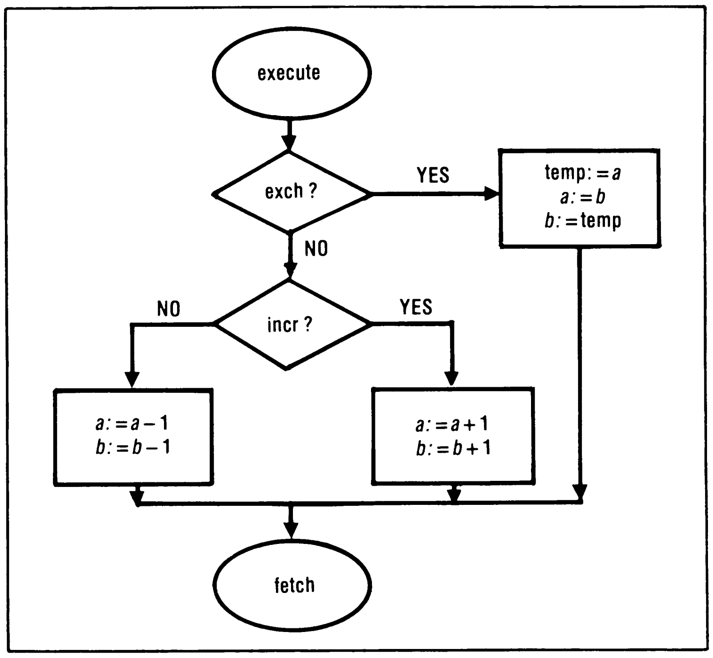
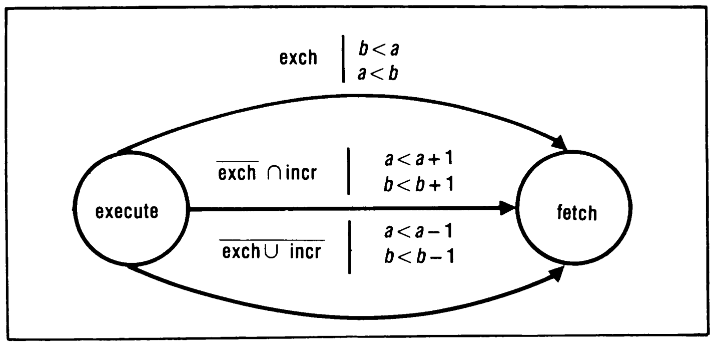


Figure 8. Flowchart of the exchange, increment, or decrement code. Note the awkwardness in describing the exchange due to the implications of the sequence of execution.



exch | b<a
a<b
$\overline{\text{exch}} \cap$ incr | a<a+1
b<b+1
execute
fetch
$\overline{\text{exch} \cup \text{incr}}$ | a<a−1
b<b−1

Figure 9. FSM implementation for exchange, increment, or decrement code. Note that FSM concept implies that the actions performed on the selected state transfer arm are simultaneous.

By inspecting the FSM representation, it is clear that actions on different "arms" of the graph can never be executed simultaneously. Therefore, the physical units executing the operations on any arm can be shared with those used for other arms. The only requirement is that the appropriate input signals be switched (the task of the multiplexers and their control) into the proper units for the selected arm. The MacPitts compiler keeps track of what units are available and generates the multiplexing and control required for this "merging."

The interpreter. A MacPitts program is not only an IC specification, it is also an algorithm specification. The MacPitts compiler generates IC layout; however, there is also an interpreter that executes the specification program on a general-purpose computer. This interpreter is invaluable because, unlike software or even wire-wrap hardware boards, it is very difficult to interactively probe and modify the internals of an integrated circuit. By using the same language to drive both the interpreter and integrated circuit compiler, human error is necessarily reduced. The algorithmic MacPitts specification is also more suitable than a structural specification for interactive function design and check.

MacPitts examples. Because of the power of MacPitts, it is very difficult to produce simple examples that demonstrate all of the features of this compiler. Many desired algorithms can be described using only one state! For example, the digital filter presented in the section on First can, in MacPitts, be described in only one state that loops back to itself forever. In fact, all such filterlike systems can be so described, and MacPitts has an *always* construct semantically equivalent to such an FSM, or *process*, as we usually call it. Another powerful feature not often used in simple examples is the ability to specify multiple parallel FSMs or *processes*.

Magnitude approximation.* The following algorithm approximates $\sqrt{a^2 + b^2}$:

$$\text{let: } g = \max \begin{vmatrix} a \\ b \end{vmatrix}$$

$$l = \min \begin{vmatrix} a \\ b \end{vmatrix}$$

$$\sqrt{a^2 + b^2} \approx \max \left\{ \begin{matrix} g \\ \frac{7}{8} g + \frac{1}{2} l \end{matrix} \right\}$$

The MacPitts code to implement this algorithm is

```
 (always
; set a-absolute b-absolute
  (cond ((bit msb a) (setq aab (    0 a)))
        (t (setq aab a)))
  (cond ((bit msb b) (setq bab ( - 0 b)))
        (t (setq bab b)))
; set g and l
  (cond ((unsigned -> aab bab)
         (setq g aab)
         (setq l bab))
        (t
         (setq l aab)
         (setq g bab)))
; sqs: = 7/8 g + 1/2 l
  (setq sqs ( + ( - g (3>> f g))
                (>> l)))
; max of sqs and g
  (cond ((unsigned ->g sqs) (setq res g))
        (t (setq res sqs)))))
```

*I am indebted to Peter Denyer for recently suggesting this algorithm, since it points up the power of the *cond* construction. The algorithm itself is much older and has a long history at MIT's Lincoln Laboratory.

The first code section computes the absolute values of *a* and *b*. The second uses these to compute *g* and *l*. Next, 7/8 *g* + 1/2 *l* is calculated and compared to *g*. Both *a* and *b* must be specified as input ports and *res* as the output port.

Note, too, that the word size must be defined and the pinout specified. These can be easily changed to accommodate modified requirements. The definitions are located in an initial MacPitts definitions section, shown below for a four-bit word size:

```
(program genmag 4
   (def msb constant 3)
   (def 1 ground)    ; pin number one is ground
   (def a port input (2 3 4 5))  ; pins 2-5 are reserved
                                 ; for input a
   (def b port input (6 7 8 9))
   (def res port output (10 11 12 13))
   (def 14 power)
   (def 15 phia)      ; other than for pin definition of
                        these pins,
   (def 16 phib)      ; clocking is implicit in the state
                        concept,
   (def 17 phic)      ; and can otherwise be ignored.
   (always
;  set a-absolute b-absolute
   . . . . . . .
```

The above code does not fully specify the design; the degree of parallelism and pipelining must still be decided. It is a simple matter to specify and modify these characteristics, and we will now investigate some of the possibilities.

If the integers *aab, bab, g, l,* and *sqs* are declared as internal *ports,* then the entire computation will proceed asynchronously and combinationally. We make the declaration in the *def* section of the MacPitts code:

```
(program magasync 4
   (def msb constant 3)
   (def aab port internal)
   (def bab port internal)
   (def g port internal)
   (def l port internal)
   (def sqs port internal)
   (def 1 ground)
   . . . . . .
```

There is really no "state" or cycle here. Rather, the program accepts inputs and produces the outputs. The combinational delay, however, is fairly large, as the inputs must propagate through several cascaded functional operator units.

For slightly greater area, pipelining can be accommodated. For example, *aab, bab, g,* and *l* can be declared as registers instead of ports. If so, the start of the program now looks like the following:

```
(program magpipe 4
  (def msb constant 3)
  (def aab register)
  (def bab register)
  (def g register)
  (def l register)
  (def sqs port internal)
  (def 1 ground)
  . . . . . .
```

With no other change in the code, this becomes a fully pipelined system. The worst-case combinational delay per cycle is shortened. It takes three cycles to obtain a result (latency from inputs to output). However, new inputs can be accommodated each cycle, thus increasing overall throughput.

Finally, by removing the parallelizing *always*, a true multistate sequential loop can be easily formed. Though slower, this design is able to share (time multiplex) some of the actual physical operation units—the number of subtraction units can be reduced from 3 to 1, for example. The determination of which physical operation units can be multiplexed is done automatically, as is the actual multiplexer specification and layout. However, register sharing is not done automatically. In this case, the *aab* and *g* values can share one register, while *bab, l,* and *sqs* share another. An extra input signal, *reset*, must be provided. The MacPitts compiler automatically generates hardware to set the process state to the beginning of the loop if the *reset* signal is high. This ensures that the resulting FSM can be started in a known state. Putting together the definition section and the sequential loop, the code to do all of this now looks like this:

```
(program magseq4
  (def msb constant 3)
  (def aab-g register)
  (def bab-l-sqs register)
  (def 1 ground)
  (def a port input (2 3 4 5))
  (def b port input (6 7 8 9))
  (def res port output (10 11 12 13))
  (def reset signal input 14)
  (def 18 power)
  (def 15 phia)
  (def 16 phib)
  (def 17 phic)
  (process compmag 0
; set a-absolute b-absolute
loop
   (cond ((bit msb a) (setq aab-g ( - 0 a)))
         (t (setq aab-g a)))
   (cond ((bit msb b) (setq bab-l-sqs ( - 0 b)))
         (t (setq bab-l-sqs b)))
; set g and l
   (cond ((not ( unsigned ->aab-g bab-l-sqs))
          (setq aab-g bab-l-sqs)
          (setq bab-l-sqs aab-g)))
; sqs: = 7/8 g + 1/2 l
   (setq bab-l-sqs (+ (- aab-g (3>> f aab-g))
              (>> bab-l-sqs)))
; max of sqs and g
   (cond ((unsigned -> aab-g bab-l-sqs)
          (setq res aab-g))
         (t (setq res bab-l-sqs)))
   (go loop)))
```

These modifications are simple ways of achieving performance goals either for area or speed. The area requirements and number of transistors for each of the three designs (four-micron minimum feature size) is shown in Table 1. Several recently proposed silicon compilers, as well as Agre's work, attempt to automate this process by using resource and timing constraints to guide an extraction of parallelism from a single, fundamentally sequential program representation.

It is also possible to construct custom designed units. For example, a max-min organelle could be created to generate *g* and *l*. This was the solution forced on First. With *cond*, we can compose functions from the primitive library set in minutes (paying area and speed penalties, of course). Without such a construct, however, circuit design and layout must be performed for many new applications.

MacPitts: past, present, and future

Several MacPitts chips have been fabricated and tested. The latest and largest of these is an automatic gain control chip, shown in Figure 10, that is actually in use in a digital vocoder system built at MIT's Lincoln Laboratory.[13] This chip was designed in a few weeks by an engineer with no previous experience in IC design. The large areas of unused space and other layout inefficiencies are continually being reduced by compiler improvements. Note that *the chip does not need to be redesigned to take advantage of any such improvements.*

A more ambitious project, a test controller, is currently being fabricated. Since the original design, which used standard functional units, was initially too large to be fabricated, some improvements to the basic organelles' layouts had to be implemented. Doing this did not require a new MacPitts specification for the test controller, and other designs could benefit from these improvements as well. However, these improvements were still not sufficient. At this point, the designer was confronted

Table 1.
Area requirements and number of transistors for each of three designs.

DESIGN	SIZE (mm × mm)	NO. OF TRANSISTORS
asynchronous	5.1 × 2.1	867
pipelined	5.4 × 2.1	1088
sequential	3.4 × 2.4	806

with three courses of action. First, a speed/area trade-off could be investigated, but this approach was deemed unsuitable for a test controller. A second method was to partition the system into several integrated circuits. MacPitts does not have any facilities for aiding the partitioning of a too-large system into a chip set. By way of comparison, First can easily partition systems because of its bit-serial approach; splitting a word-parallel design usually generates overly large pin counts. The third approach, the one finally used, was to create some special-purpose organelles.

These designs, and others, have pointed out several weaknesses of MacPitts. Performance prediction and automatic trade-off analysis is not a strong point; MacPitts lacks system partitioning facilities and only generates NMOS designs. Finally, the test generation mechanisms of MacPitts still require human interaction. We have no shortage of ideas, but a great deal of work will be necessary to overcome these problems. Nevertheless, MacPitts has demonstrated its fundamental goal: Custom integrated circuits can be effectively designed from an algorithmic specification of the circuits' behavior. ■

Acknowledgments

I thank my colleagues Jeffrey M. Siskind and Kenneth W. Crouch, who, together with the author, invented, developed, and implemented MacPitts. I would also like to thank Allan Anderson and Peter Blankenship for their help in editing this paper.

This work was sponsored by the Defense Advanced Research Projects Agency. The United States government assumes no responsibility for the information presented.

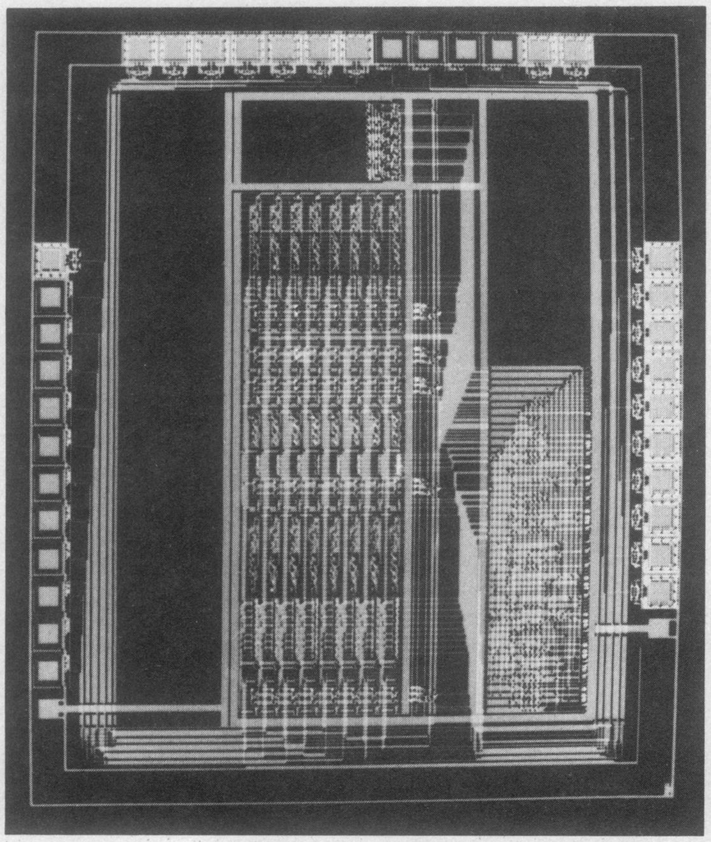
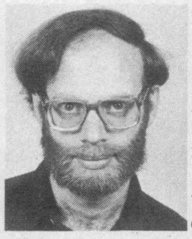


Figure 10. Automatic gain control chip.

References

1. J. Werner, "The Silicon Compiler: Panacea, Wishful Thinking, or Old Hat?," *VLSI Design*, Vol. 3, No. 5, Sept./Oct. 1982, pp. 46-52.
2. Gordon Moore, "VLSI: Some Fundamental Challenges," *IEEE Spectrum*, Vol. 16, No. 4, Apr. 1979, pp. 30-37.
3. Rex Rice, *VLSI: The Coming Revolution in Applications and Design,* IEEE CS Press, Los Alamitos, Calif., 1980.
4. Jeffrey Siskind, Jay Southard, and Kenneth Crouch, "Generating Custom High-Performance VLSI Designs from Succinct Algorithmic Descriptions," *Proc. Conf. Advanced Research in VLSI,* P. Penfield, ed., MIT, Cambridge, Mass., Jan. 1982, pp. 28-40.
5. *Proc. IEEE* (special issue on VLSI design), Vol. 71, No. 1, Jan. 1983.
6. Borgini et al., *Automated Design Procedures for VLSI,* final report DELET-TR-78-2960-F, Advanced Technical Laboratory, RCA Government Systems Division, 1981.
7. J. D. Williams, "Sticks—A Graphical Compiler for High-Level LSI Design, *AFIPS Conf. Proc.*, Vol. 47, 1978 NCC, pp. 289-295.
8. C. Mead and L. Conway, *Introduction to VLSI Systems,* Addison-Wesley, Reading, Mass., 1980.
9. D. Johannsen, "Bristle Blocks: A Silicon Compiler," *Proc. 16th Design Automation Conf.*, June 1979, pp. 310-313.
10. H. E. Shrobe, "The Data Path Generator," *Proc. Conf. Advanced Research in VLSI,* P. Penfield, ed., MIT, Cambridge, Mass., Jan. 1982, pp. 175-181
11. P. E. Agre, "Designing a High-Level Silicon Compiler," *VLSI Research Review*, MIT, Cambridge, Mass., Dec. 1982.
12. Neil Bergman, " A Case Study of the F.I.R.S.T. Silicon Compiler," *Proc. Third CalTech Conf. VLSI,* R. Bryant, ed., Pasadena, Calif., Mar. 1983, pp. 413-430.
13. J. A. Feldman and E. J. Beauchemin, " A Custom IC for Automatic Gain Control in LPC Vocoders," *Proc. ICASSP 83*, Boston, Mass., Apr. 1983.



Jay R. Southard currently works for Metalogic, Inc., Cambridge, Mass. From 1979 to June 1983, he was a member of the MIT Lincoln Laboratory, where he engaged in research in VLSI. Southard received a BA in mathematics and physics in 1973 from Grinnell College, Grinnell, Iowa, and an MSEE from Stanford University in 1974. He has also attended courses at Adelphi University and MIT. Southard is a member of Phi Beta Kappa.

Southard's address is Metalogic Inc., 725 Concord Ave., Cambridge, Mass., 02138.