GENERATING CUSTOM HIGH PERFORMANCE VLSI DESIGNS FROM SUCCINCT ALGORITHMIC DESCRIPTIONS*

Jeffrey Mark Siskind
Jay Roger Southard
Kenneth Walter Crouch

MIT Lincoln Laboratory
P. O. Box 73
Lexington MA 02173

ABSTRACT

An integrated circuit design may be specified at different levels of detail ranging from low level geometric masks to high level algorithms. The design process can be viewed as the successive refinement of higher level descriptions into lower level ones. Traditionally this refinement process has been performed by hand and has been exceedingly tedious, time consuming, inflexible, and error prone. Many designs fit into the framework of microprogram sequenced data path operations. For this large class of systems, this paper discusses an alternative to hand crafted design.

A flexible register transfer type language known as MacPitts was developed particularly suited to the capabilities available through custom VLSI design. Features of this language include multiple way branching, nested conditions of arbitrary boolean expressions, subroutine capability, and parallel processes. A compiler was developed which converts programs in this language directly to mask level specifications. The target architecture for implementing the system is a combination of finite state machines, one for each of the parallel processes in the source code, and a data path unit. The compiler consists of two levels of routines. The higher level routines examine the source code and extract a technology independent intermediate level description of the system in terms of data path specifications, control equations, and state assignments. The lower level routines bind this intermediate level description to an actual mask layout. The output of the lower level routines is actual CIF code for the MOSIS/NMOS process.

*This work was performed at the M.I.T Lincoln Laboratory as part of the Restructurable VLSI research program sponsored by the Information Processing Techniques Office of the Defense Advanced Research Projects Agency. The United Stated Government assumes no responsibility for the information presented.

Introduction

The design of an integrated digital system can be viewed as a refinement process. The designer starts with an abstract concept of the desired functionality of the target system and refines this eventually to a geometric description of rectangles for a mask set. Throughout the design process, the designer makes use of several intermediate level descriptions. These include an algorithmic method for achieving the desired functionality, an architecture for performing that algorithm, and the implementation of that architecture as a logic diagram. Before the advent of VLSI, digital system designers refined their designs to a gate or IC package level specification. Design of VLSI systems is complicated by the additional refinement needed to layout the geometry associated with a gate level description. This complexity makes the design of custom VLSI circuits time consuming, tedious, inflexible, error prone, and expensive.

VLSI offers the potential for fabricating increasingly larger and more complex systems. Accordingly, there is a desire to implement larger portions of systems, or even entire systems, as custom VLSI circuits to take advantage of this potential. Due to the additional design expense however, it is not always cost effective to choose custom VLSI over discrete logic as the method for implementing a system. A reduction in the

complexity of integrated circuit design is needed to realize systems of the magnitude attainable with VLSI.

Previous approaches to decreasing design complexity have centered around the lowest level of refinement from a logic diagram to a mask layout. Design methodologies based on increased geometric regularity have been suggested as ways of improving designer productivity. Various CAD tools have been developed which aid the designer in performing low level circuit layout. All of these approaches fail to decrease the complexity of integrated circuit design enough to make custom VLSI preferable to discrete logic in many cases. The reason for this is that present methods concentrate only on the low level layout task which by its very nature is an additional step beyond logic design required only when designing custom integrated circuits. By broadening the scope of the refinement automation to accept an algorithmic description as input, the design time for custom integrated circuits can be made shorter than even traditional digital system design.

This paper discusses a language called MacPitts* which can specify circuits at the algorithmic level.
A compiler is under development which refines this description into an NMOS circuit layout. Part of the compiler is technology independent however. This part extracts an intermediate level description of the circuit under design. The rest of the compiler uses this description to synthesize a circuit layout. The output of these layout routines at present is CIF, which can be used to fabricate the circuit using the ARPA/MOSIS facility. The MacPitts design system includes a functional simulator. This program simulates the logical behavior of the circuit under design and can aid in verifying the design's correctness before fabrication. The functional simulator is driven by the intermediate level description output from the compiler's technology independent component. This alleviates the necessity for instantiating either the circuit or the geometry before simulation. The remainder of this paper will discuss the target architecture of circuits generated by the compiler, some interesting features of the source language, the compiler implementation, and the implication of this method on the VLSI design process.

*The name MacPitts is derived from a combination of the names McCulloch and Pitts[1], the originators of the study of neurological systems from a mathematical and logical standpoint.

Design Languages and Target Architectures

The two most important decisions made when designing the compiler were the choice of an appropriate design language for describing the algorithm performed by a circuit, and a target architecture suitable for implementing designs specified in that language. The design language must be general enough to describe a sufficiently large class of systems. It must be able to specify designs in this class concisely and allow for an efficient implementation in the target architecture. This target architecture must be general enough to allow an efficient implementation of all designs representable in the source design language. It must also be sufficiently constrained to allow automatic layout synthesis. There is no known way for categorizing the alternatives available with arbitrary mask layout. This makes the general mask layout task difficult to automate. Diversity in this large design space must be limited by reducing the number of degrees of freedom and restricting layout to a fixed target architecture.

Our objective was to choose a design language suitable for specifying signal processing and similar high throughput systems. These systems often require a high degree of concurrency to achieve the required throughput. The MacPitts language therefore, allows explicit specification of parallel computation to be performed by the circuit. A register transfer type language with a unique parallel semantics is used to specify the parallel computations. The target architecture chosen supports the concurrent computation which can be specified in the design language. This architecture is depicted in figure 1. Within such an architectural framework, a digital system is partitioned into two distinct sections, a data-path and a control unit. The data-path consists of an assemblage of registers of a fixed width specified by the MacPitts source program, as well as operators for modifying and testing the data stored in those registers. Data is communicated between the data-path and the external world through *ports*. Ports are parallel buses of wires which have the same width as registers in the data-path. A given port may be declared as being an *input*

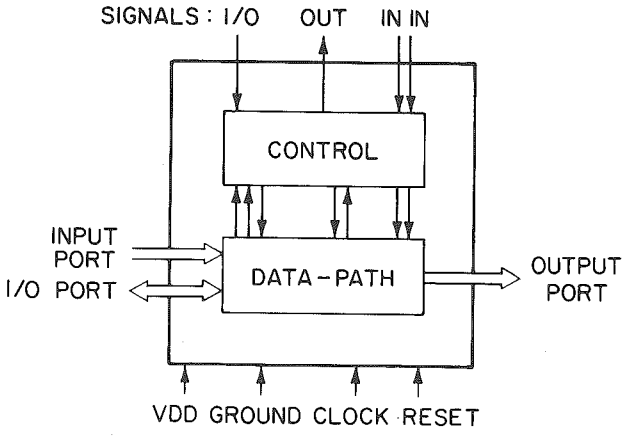

Figure 1 - MacPitts Target Architecture

port, a *tri-state* output port, or an *i/o* port. The sequence of operations performed by the data-path is governed by the control unit. Control inputs from the control unit to the data-path determine which operations occur in the data-path during a given clock cycle. Test outputs from the data-path returning to the control unit allow the sequence of operations performed by the control unit to vary depending on the data stored in the registers. The control sequence may also depend on external signals input to the control unit. The control unit communicates with the external world using *signals* in much the same way as the data-path uses ports. A given signal is a single wire and may be declared as *input*, *output*, *tri-state* output, or *i/o* in the algorithmic description. In addition to the pins dedicated to the ports and signals which vary from system to system, every design will have power, ground, clock, and reset pins.

The MacPitts Design Language

As the MacPitts compiler is implemented in LISP, the syntax of the MacPitts design language is motivated by the LISP parenthesized notation. Using this notation simplified the design of the compiler by eliminating the need for a parser. A slightly abridged grammar for MacPitts is given in the appendix. Although the design language resembles LISP in many ways, its semantics is quite different. MacPitts programs are written in a state oriented register transfer fashion which differs from the applicative and recursive style of LISP.

A MacPitts program consists of a set of processes all of which are performed in parallel. Each process is divided into states which are executed sequentially. The states of each process are disjoint from those of other processes allowing the hardware to perform the control structure and actions of each process independently of the state of other processes. The operations performed by a process in a given state are specified by a form. Each top level form corresponds to a single machine state and is executed in one clock cycle. A state may be given a name by preceding the form with a label. Normally, execution proceeds sequentially from one state to the following state at each clock pulse. A *go* form can be used however, to deviate from this sequential flow by causing the named state to be executed next rather than the syntactically following state. A group of states can be called as a subroutine using the *call* form. At the end of the subroutine control can be returned to the statement following the call form using a *return* form. The MacPitts compiler will construct a control unit which can implement the combined control structures of all of the processes in a program. The control unit architecture supports sequential program flow as well as nested subroutine calls. The compiler will use restricted versions of this architecture if some features such as subroutines are not needed.

Operations performed by the data-path during a given state are specified with a *setq* form. The setq form causes the data-path to evaluate a sequence of operations on either input data or on data stored in registers. The result of these operations is either loaded into a destination register at the end of the clock cycle or can be output from the circuit through a port. The compiler includes enough copies of each operator in the data-path so that separate processes do not conflict over the attempted shared use of a single resource. The data-path can cascade several operations together using parallel local buses which are described in the target architecture section. This allows forms such as the following to be executed in a single clock cycle.

```
;   a := b-c using twos complement arithmetic
(setq a (+ b (1+ (not c))))
```

The *cond* form allows the conditional execution of other forms during a given state. It consists of a list of guards only one of which is to be executed. Each guard begins with a condition which determines whether the remaining forms in the guard are to be executed. The first guard whose condition is true enables the execution of the forms following the condition in that guard. This is illustrated by the following example.

```
(cond (condition1
       (cond (condition2 form1 form2)
             (condition3 form3 form4 form5)
             (t form6)))
      (condition4
       (cond (condition5 form7 form8))
       (cond (condition6 form9))
       form10))
```

If condition1 is false and condition4 is true then form10 is executed. If condition5 is true then form7 and form8 are executed along with form10. Likewise if condition6 is true then form9 is executed in parallel as well.

Conditions are derived from either external input signals, signals from other processes, or tests performed by the data-path. More complex conditions can be constructed from these primitive ones using the logical operators *and*, *or*, and *not* to build arbitrary boolean expressions. Expressions are not limited to sum-of-products notation used by PLA based finite state machine compilers such as SLIM[2].

The semantics of the cond statement is inherently parallel. It is one of the most powerful features for providing high performance designs. The conditions of the alternative guards are checked in parallel. Likewise, all forms in the guard which is enabled are executed simultaneously in one clock cycle. The compiler makes the conditions of different guards in one cond form mutually exclusive and implements them using combinatorial logic in the control unit. This logic is used to enable or to inhibit the execution of the forms controlled by that guard in parallel. Often, a restricted form of cond is used to enable parallel execution of several forms during one clock cycle, without being dependent upon some condition.

```
(cond (t form1 form2 form3 ...))
```

Since the above form is so widely used, the following macro abbreviation has been provided for it.

```
(par form1 form2 form3 ...)
```

This illustrates the use of the convenient macro expansion feature of the compiler for extending the MacPitts syntax.

When several forms which access a common register are executed in parallel, a question arises as to the semantics of register loads performed by setq. The semantics of MacPitts states that during a state which is executed in a single clock cycle, all registers are read before they are written. The compiler generates master slave registers which support this semantic feature. This allows operations similar to swapping registers to be correctly performed in one clock cycle.

```
(par (setq a b)  ;swap the contents of registers
     (setq b a)) ;a and b in one clock cycle
```

It is a semantic error to attempt to load a register simultaneously from different sources. The functional simulator will detect this and other semantic errors to help debug and verify a design before fabrication. Separate processes which communicate using shared registers can be synchronized using signals to guarantee that these errors don't occur. Signals need not be defined as input or output in which case they are assumed to used for internal communication between processes. One process can assert a signal using the setq form while another process checks that signal as a condition in a cond form. The following example illustrates the use of signals in this way.

```
;presumably some huge signal processing task,
; typical at Lincoln Lab
     (program signal-processor
      (process filter 0
       ... part of filter program ...
;filter needs multiplication
;load multiplier registers
       (par (setq multiplier a)
            (setq multiplicand b)
            (setq start-multiply t))
;signal multiplier to start,
; and wait for completion
```

```
wait   (cond ((not multiply-finished) (go wait))
;before accessing result
             (t (setq c product)))
;and continuing with filter
       ... rest of filter program ...)

;multiplier - wait for multiply request,
;          - and notifiy when done
      (process multiplier 0
wait   (cond ((not start-multiply) (go wait)))
       ... code for multiplication algorithm ...
       (setq multiply-finished t)))
```

The following is an example of a complete MacPitts program. The program implements a taxicab meter. It demonstrates the use of two processes to perform simultaneous tasks and the use of signals to communicate between those tasks. It demonstrates a system which can not be efficiently implemented as a simple finite state machine and needs the partitioning into a control unit and separate data path.

```
(program taxi-cab-meter

 (def 8 word-length)
 (def timer register)
 (def fare register)
 (def display port tri-state (1 2 3 4 5 6 7 8))
 (def time-on signal input 9)
 (def hire signal input 10)
 (def mile-mark signal input 11)
 (def maximum-time constant 100)
 (def base-fare constant 190)
 (def cost-per-mile constant 50)
 (def cost-per-time constant 10)

 (process time-clock 0
  off
;when driver turns on time clock
;    -then- clear timer and start counting
;    -else- hold timer
     (cond (time-on (setq timer 0)
                    (go on))
           (t       (go off)))
  on
;when timer overflows
;    -then- reset timer and signal fare-clock
;    -else- just count up
     (cond (time-on
              (cond ((= timer maximum-time)
                     (setq timer 0)
                     (setq charge-time t))
                    (t (setq timer
                             (1+ timer))))
              (go on))
;if driver turns off time clock
;   -then- reset it and return to off state
           (t  (setq timer 0)
               (go off))))

 (process fare-clock 0
  for-hire
;if taxi is hired
;  -then- start with base fare
;  -else- wait for hire
     (cond (hire (setq fare base-fare)
                 (go hired))
           (t    (go for-hire)))
  hired
;charge for time and/or distance, depending
;       on flags
;until end of fare
     (cond
       (t (cond
            ((not hire)
             (go for-hire))
            ((and charge-time mile-mark)
             (setq fare (+ (+ fare cost-per-mile)
                           cost-per-time)))
            (charge-time
             (setq fare (+ fare cost-per-time)))
            (mile-mark
             (setq fare (+ fare cost-per-mile))))
          (setq display fare)
          (go hired)))))
```

This design contains two processes, one to implement the time clock and one for the distance clock. The time clock may optionally be turned off by the signal time-on from driver. If running, the time clock signals the fare clock on overflow. The fare clock keeps track of the combined fare which is based both on time and distance. It is enabled by the hire signal from the driver. When first detecting the hire signal, the fare clock is reset to the base fare. The external mile-mark signal comes from the odometer to signal the incrementation of the fare-clock. The fare is continuously displayed through an output port.

The MacPitts Target Architecture

As was mentioned previously, the MacPitts compiler designs circuits composed of data-paths and control units. This section will discuss the organization of these components in greater detail.

The conventional way of designing a data-path is to partition it into a separate register array and ALU as shown in figure 2. This approach is typical of many VLSI

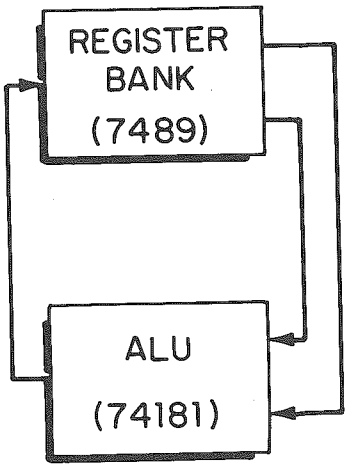

Figure 2 - Conventional Data-Path Organization

microprocessors such as RISC[3] and the Hewlett-Packard 32 bit machine[4]. The drawback of this organization is that it does not allow for the parallel data-path operations need by the design language for achieving high throughput for signal processing applications. A scheme is needed whereby multiple operations can be performed during a single clock cycle rather than using a complete read/modify/write cycle along a single bus for each operation. Such a scheme is given in figure 3. Interspersed among the registers are as many functional units as are needed to implement the parallel operations required by the algorithm. Each functional unit need not be a complete ALU and may perform either a single dedicated or a small set of functions and tests. Functional units such as adders, incrementers, and shifters compute the values of formulas used in setq forms in the MacPitts design. The output of function units is a full word used by the data-path. Test units such as comparators provide output test results to the control unit to implement the primitive conditions used in the cond form. A unit such as an adder can provide

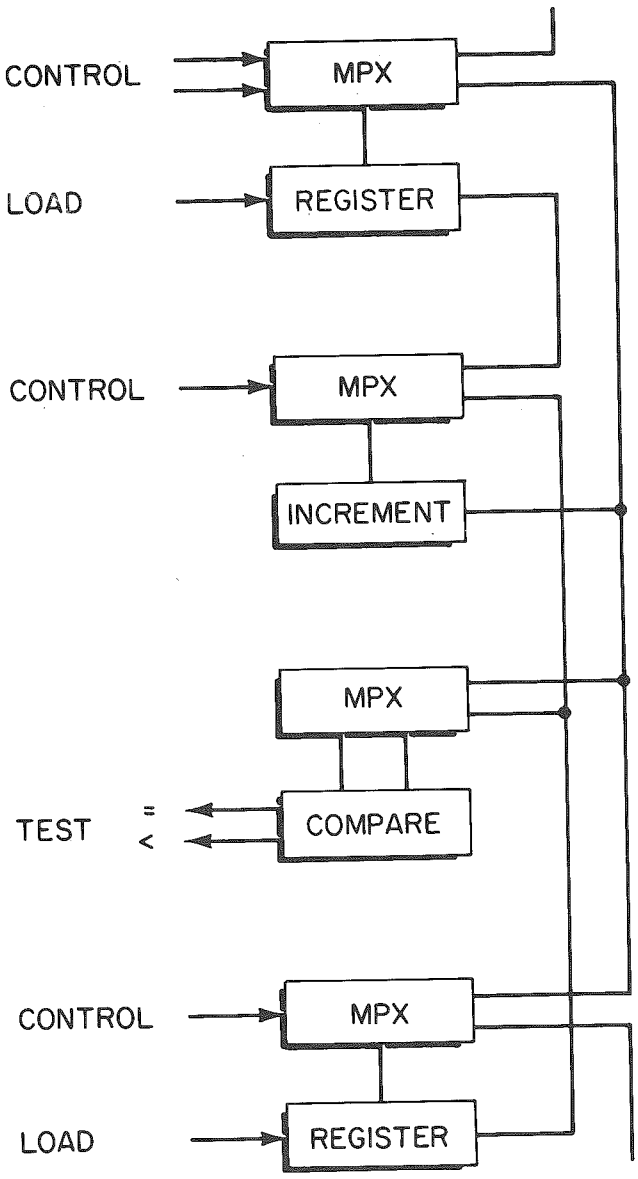

Figure 3 - Organization of MacPitts Data-Path

both function and test output. The sum output is a function output to be used further by the data-path. The overflow output is a test signal which is used by the control.

The quantity and type of units required in the data-path will vary from system to system. The MacPitts compiler examines the source program and extracts the necessary information to construct a custom data-path suited to the application. All of the units are

interconnected by dedicated local buses as required by the operations performed in the MacPitts design. The availability of separate parallel buses allows the different units to operate simultaneously. Often a given unit, usually a register, will need to take its inputs from a variety of different sources. Therefore each unit is provided with a multiplexer at its inputs to choose from which local bus to take its inputs during a given clock cycle. The multiplexer can also present the unit with a given constant value as input instead of the data from a local bus if desired. The control lines which determine the selection of different multiplexer alternatives as well as control lines to the units themselves allow the control unit to determine the sequence of operations performed by the data-path.

A more detailed picture of the layout scheme for the data-path is given in figure 4. This method is similar to the one proposed by Suzim[5]. The data-path is constructed as a rectangular array of primitive components termed *organelles*. Each organelle is a single bit slice of a given unit such as a register or an adder. Along a vertical slice of the data-path, organelles are stacked to form units. Each horizontal slice corresponds to a single bit slice of each unit in the data-path. The local interconnections between units are partitioned so that each bit slice of interconnection lies in the horizontal channel between two bit slices of organelles. These buses are segmented so that a given bus need not extend to both edges of the data-path. Instead, several short local buses can share a single collinear track. The routine which generates the layout of the data-path will optimally pack the local buses to minimize channel width.

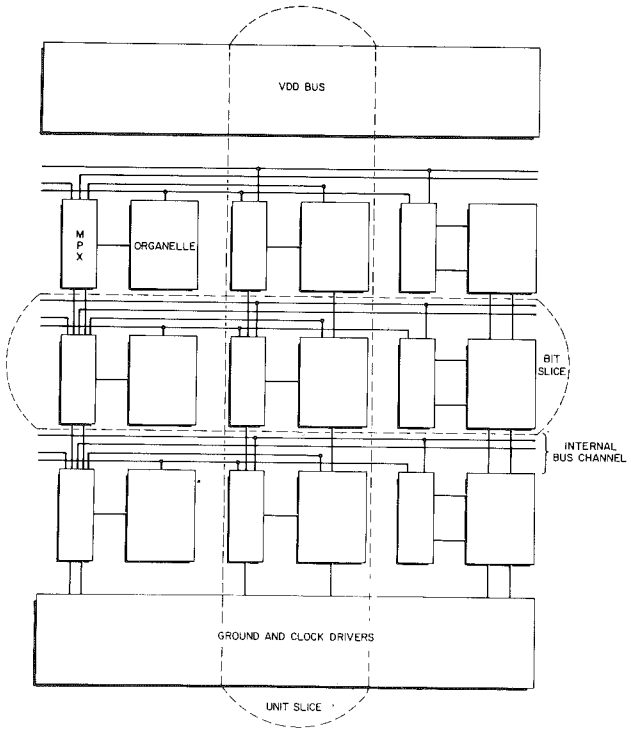

Figure 4 - Detailed Organization of MacPitts Data-Paths

The layout of each organelle is generated by a LISP function called by the compiler. These functions all have standard calling sequences and return data in a well defined format. The system contains a library of standard organelles such as adders, incrementers, shifters, and registers. This library may be augmented with new organelle definitions to provide new function and test unit types. The syntax of the MacPitts design language is easily extended to accommodate these new units.

Organelles for different units vary in their dimension, interconnection and power requirements. The organelle layout functions can be called with various query requests to obtain this information. When constructing the data-path, MacPitts uses this information to properly space the organelles and to stretch their connection points to connect with the control lines, power lines, and local interconnection buses. Custom multiplexers are appended to the inputs of each organelle to select appropriate constant and local bus data. The compiler automatically sizes the power and ground buses of the data-path based on the power requirements of the organelles as well. Figure 5 is a stipple plot showing the stretch points and other query information for the register bit organelle which is a typical organelle used by the system.

Since the layout of an organelle is generated by a LISP function, more complicated parameterized organelles are possible. This would allow an organelles' structure to vary with bit position and word length and can be used to build more complicated units such as carry look-ahead

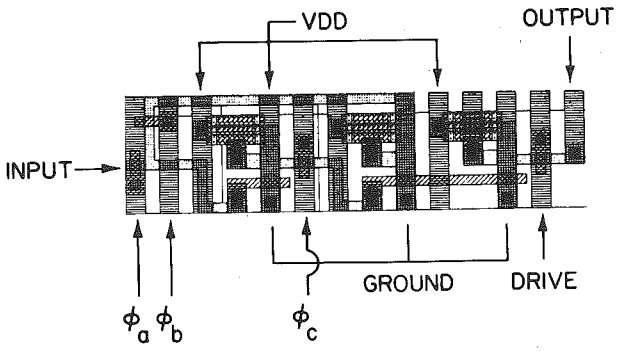

Figure 5 - A Sample Register Bit Organelle

adders and multiple bit barrel shifters.

The control unit is implemented as a variant of a finite state machine. Typical finite state machines are composed of combinatorial logic with an attached state register. The logic computes not only the output signals to be generated but also the next state of the machine. Most MacPitts programs have a very sequential flow of control states. Using logic to implement the next state equations is very inefficient when they perform a simple increment function. In this case, the amount of combinatorial logic can be reduced by replacing the simple state register with a counter. Now, the logic need generate next state information only when the machine deviates from sequential program flow. The simple counter can be extended with a push down stack to store temporary states. This implements the subroutine call and return constructs of MacPitts. This combination of a counter with a state stack is termed a sequencer. The complete control unit as is generated by MacPitts is composed of combinatorial logic connected with possibly several sequencers. This architecture which is depicted in figure 6 allows several independent and simultaneous control processes to execute together as required to support the MacPitts syntax and semantics. Partitioning a MacPitts program into several processes reduces the size of the control unit by

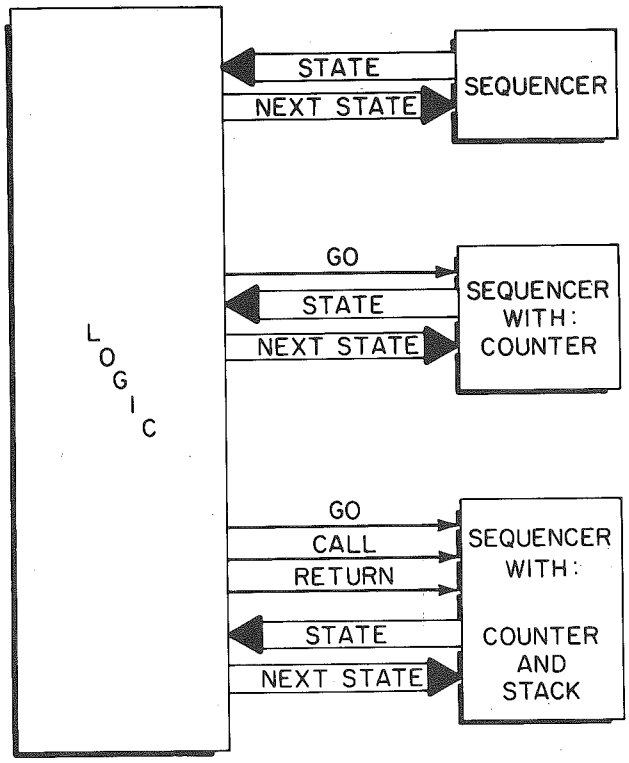

Figure 6 - Control Unit Architecture

decomposing a large finite state machine into the cross product of several smaller ones. The sequencers themselves can be thought of as small data-paths of the type described above. The stack and counter of a sequencer can be implemented as a set of registers along with an incrementer which can allow parallel transfers and operations. The following MacPitts code describes the semantics of a sequencer which is four bits wide and has a stack depth of three. The program makes use of the *always* construct. The always construct allows specifying a stateless process which is to be continuously executed. Such a program results in a layout which contains a data-path and control logic which does not depend upon any sequencer inputs. Therefore such a design is not self referential.

```
(program sequencer
 (def 4 word-length)
 (def state port tri-state (1 2 3 4))
 (def next-state port input (5 6 7 8))
 (def go signal input 9)
 (def call signal input 10)
 (def return signal input 11)
 (def s1 register)
 (def s2 register)
 (def s3 register)
 (always (cond (go
;load state register
                    (setq s1 next-state))
              (call
;load state register and push stack
                    (setq s1 next-state)
                    (setq s2 (1+ s1))
                    (setq s3 s2))
              (return
;pop stack
                    (setq s1 s2)
                    (setq s2 s3))
;if no signal go to next sate
              (t     (setq s1 (1+ s1))))
;and always output present state
        (setq state s1)))
```

The combinatorial logic portion of the control unit is presently implemented by MacPitts in the form of a Weinberger style gate array[6] This is essentially a restricted architecture for laying out a random collection of arbitrary input size NOR gates. It is similar to a PLA in that combinatorial logic is synthesized from NOR gates. A PLA is restricted however, to contain only two levels of NOR gates, the AND plane and the OR plane. The Weinberger method allows unbalanced NOR gate graphs of arbitrary depth to be configured. This allows an efficient implementation of the control logic required for arbitrary boolean conditions allowed in a MacPitts program. An attempt to normalize the conditions found in typical MacPitts programs into the sum-of-products notation required for a PLA has shown that the Weinberger array should be more compact than a PLA when nested boolean conditions are used. In the Weinberger style, NOR gates are arranged as vertical columns of an array. Each gate has two lines, an output line and a pull down line, connected to a pull up device at the top of the column. Straps are placed horizontally in the array to connect the output line of one gate to the input pull down transistors along the pull down lines of other gates. A stick diagram of a circuit designed in this fashion is given in figure 7.

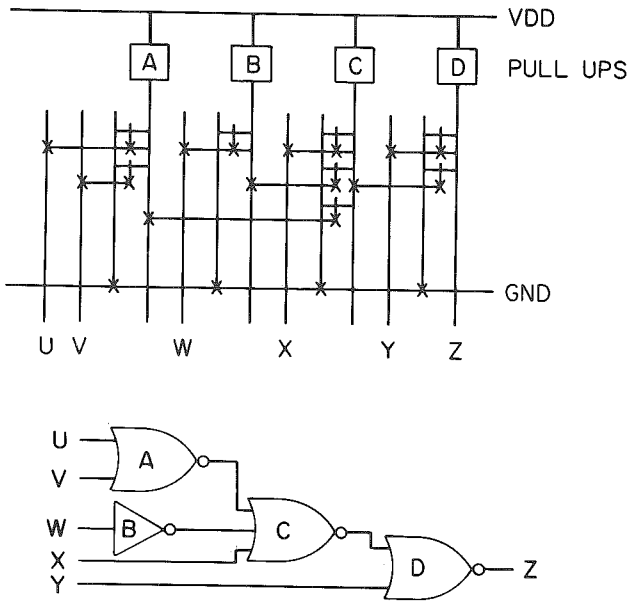

Figure 7 - Weinberger Gate Array Implementation of Random Logic

Implementation

The MacPitts compiler is implemented as a set of LISP functions which read the MacPitts source and produce CIF code for the MOSIS/NMOS process as output. The higher level routines extract from the source program an intermediate level description of the circuit to be designed. This description describes the data-path as a set of units and the control unit in terms of boolean formulas. The intermediate level description is process independent and is output by the compiler to be used for functional simulation. The lower level routines use this intermediate representation to lay out the circuit. A brief description of the major routines used by the compiler follows.

Prepass - All of the syntax checks of the compiler are combined into one set of routines. This eases the task of writing and verifying the rest of the compiler by allowing it to assume that it has received a syntactically correct program. This pass

processes all of the definitions given in the source program for such things as signals and ports and returns them as a definitions list in a standard format for use by other routines in the compiler. This pass also implements a macro expansion feature which makes the MacPitts language somewhat extensible.

Data Path Extraction - Scans the source code to determine the set of registers and operators needed to implement the required data path. It understands the parallel semantics of certain MacPitts constructs which allows it to optimize the data path. When segments of code have sequential or mutually exclusive execution the operators required by those code segments can be shared. Code segments which are to be executed in parallel will cause the necessary duplication of operators in the data path. The output of this pass is a technology independent data path specification in a well defined format.

Sequencer Extraction - Determines for each process the width and depth of the sequencer needed. Also determines whether a given process needs a fully general sequencer or whether it can eliminate the call/return stack and possibly the counter. This pass also builds a symbol table of go target labels and adds this to the definitions list. It outputs a specification of the width and stack depth required of the sequencers for each process in a technology independent manner.

Control Extraction - Converts the source code into boolean equations which describe the interaction between the data path, the sequencers, and the outside world. It accepts a rather general input format which allows nested conditionals, arbitrary boolean conditions and complex formulae. The output is again technology independent.

Normalizer - Used by the control extraction routines to normalize the boolean equations into NOR gate form. An attempt is made to keep the depth of the NOR gate array to a minimum but not at the expense of expanding the equations excessively into sum-of-products form. Heuristics are used to help remove unnecessary terms in the boolean equations. These heuristics are being improved with the goal of normalizing boolean formulae into more compact forms under different constraints.

Data Path Layout - Commits the data path specification previously extracted from the source program to a geometric layout. Lays out the array of organelles and stretches their connection points. Provides each organelle with a custom designed multiplexer at its input if needed. Optimally packs the local internal buses used to interconnect the units. Connects the organelles to these buses, to the control lines, and to automatically sized power and ground buses.

Sequencer Layout - Builds geometric layouts for the sequencers needed for each process in the source program. Uses the data-path layout routines as described previously.

Control Layout - Generates the layout of the combinatorial logic for the control unit using the Weinberger gate array layout style. Orders the NOR gate columns to simplify the connections required between the control unit and the data-path. Packs the straps using the same method used to pack the local internal buses in the data-path to minimize the width of the control unit.

Router - Two types of routing are performed by the MacPitts compiler. Whenever possible layouts are constrained so that river routing is possible. River routing is used to connect the multiplexers to each organelle. River routing is also used to connect the combinatorial control logic with the sequencers and the data-path. A more general routing scheme will be used to connect the major components to the I/O pads.

Organnelle Library - The Data Path Layout routine references this organelle library to determine the layout and characteristics of each primitive operator. The term organelle refers to a single bit slice of a register or operator. Organelles are coded as LISP functions which use L5 primitives to generate layouts. The organelles all have a standard calling sequence which enables the Data Path Layout routines to query the organelle and determine certain key characteristics. The Data Path Layout routine can be augmented with new operators simply by adding new definitions to the library in this well defined format. The basic initial library at present includes the organelles for and, or, not, nand, nor, xor, equ, 1+, 1-, +, -, =, <, <<, >>.

L5 - A Lisp based geometric IC layout language used by all the layout routines to construct CIF files. It is similar to LICL[7], a text based layout language implemented in C. A prime attribute of L5 is its total applicative nature.

Simulator - Performs a functional simulation of the design's logical behavior from the intermediate level technology independent code output by the compiler. It simulates the parallel execution of processes as well as the data path operations. The simulator can be extended to handle new data path operators by including a simulation form with each new organelle added to the library. Semantic error checking is provided in the simulator to aid the verification of designs before fabrication.

A New Approach to VLSI Design

The use of algorithmic level design specifications and the automation of the refinement process will change many aspects of VLSI design in addition to the layout task. This is particularly true for the design verification task. It is common practice to analyze a design in an attempt to remove any apparent flaws and guarantee a circuit's correctness prior to fabrication. For hand generated layouts this verification usually consists of checking for design rule violations, extracting a connective transistor schematic from the hand layout, and simulating the behavior of that transistor schematic using a switch level simulator. All of these are compute intensive and time consuming tasks. A typical project will require many iterations of design and verification to remove all flaws. The time required by the verification task slows down this iteration considerably. Generating designs using MacPitts will alleviate much of the verification task. Layouts generated by MacPitts are correct by synthesis obviating the need for design rule checking.

The intermediate level description output by the MacPitts compiler can be used to drive a functional simulator. This simulator emulates the behavior of data-paths, sequencers, and boolean control logic at a level far above transistor switching. As the functional simulator does not require a transistor schematic for switch level simulation, the costly step of node extraction can be eliminated. The compiler can generate the intermediate level description needed for simulation in far less time than needed to generate a layout. Functional simulation is also faster than switch level simulation and provides additional semantic information to the designer which would be lost if the design were refined to the geometric representation and then reverse engineered. The simulator interacts with the designer in a language analogous to the MacPitts source program in terms of states, registers, signals and I/O ports. The designer can examine any of these as well as change them as he cycles through the simulation of his design. The behavioral description of the data-path units required by the simulator is included as part of an organelle description in the library. As new organelles are added, they are provided with such a behavioral specification so that the simulator is augmented as well. The iteration time from design through verification and back to design is thus greatly reduced with MacPitts as compared to hand layout.

Using the MacPitts style of design promises additional benefits which need to be explored in the future. As designs are specified at a technology independent level one can conceive of compiling the same design to alternate technologies or design rules. To accomplish this only the lower level "code generation" routines of the MacPitts compiler need be altered. This would allow the same source language and even designs to take advantage of newer technologies such as buried contacts, CMOS, or multiple level metal. Additional, more far reaching benefits, are possible. As new techniques for VLSI design and silicon compilation are developed these can be incorporated into MacPitts. Possibilities include automatic test vector generation and generation of testable, self testing and fault tolerant circuits. Automation of such techniques when they are developed would be difficult if layouts continue to generated by hand. These possibilities are conceivable only when designs are specified at a high enough level of abstraction and are refined to the layout level automatically as is performed by the MacPitts system.

REFERENCES

[1] McCulloch, W.S. and Pitts, W., *A Logical Calculus of the Ideas Immanent in Nervous Activity*, Bull. Math. Biophysics, pp. 115-133, 1943.

[2] Hennessy, John , *SLIM: A Language for Microcode Description and Simulation in VLSI*, CSL Stanford University, preprint of paper, May 1981.

[3] Fitzpatrik, D.T., et. al., *VLSI Implementations of a Reduced Instruction Set Computer*, Proc. CMU Conf. on VLSI Systems and Computations, pp. 327-336. Oct. 1981.

[4] Beyers, Joseph W., et. al., *A 32b VLSI CPU Chip*, ISSCC, pp. 104-105, Feb. 1981.

[5] Suzim, Altamiro Amadeu, *Data Processing Section for Microprocessor-Like Integrated Circuits*, IEEE JSSC, vol. SC-16, pp. 233-235, June 1981.

[6] Weinberger, A., *Large Scale Integration of MOS Complex Logic: A Layout Method*, IEEE JSSC, vol.SC-2, pp.182-190, Dec. 1967.

[7] Plummer, William W., *Chip Maker's Guide*, MIT-LL memo, Sept. 1981.