Designing Gate Arrays Using a Silicon Compiler

John P. Gray, Irene Buchanan & Peter S. Robertson

Lattice Logic Ltd
6 Albany Lane, Edinburgh, EH1 3QP, Scotland.

ABSTRACT

This paper describes a programming environment in which gate array designs can be developed. It allows the engineer to design for performance, wirability and testability by manipulating a textual description of a design. The principle features of this are a high-level language for design description, completely automatic layout, and an integrated simulator. The total package can be referred to as a silicon compiler in the gate array design style.

1. INTRODUCTION

Present approaches to design automation tools for gate arrays emphasise general placement and routing algorithms for use with a range of proprietary chip images. This very often leads to inefficiencies in physical design by sacrificing active area to accommodate wiring. An alternative approach is to recognise that communication (wire) is at least as important as function and to design the chip image accordingly. Thus by allowing a wiring management strategy and physical design algorithms to dictate floor plan and cell design, it becomes possible to achieve fully automatic physical design. The realisation of such a physical design subsystem, together with an appropriate notation for hierarchical description, can be thought of as a silicon compiler.

This paper describes such a silicon compiler. With this approach chip design is seen as analogous to software development where formal notations provide a convenient method of enforcing good design practice. This analogy is explored by the description of the key components of a design environment constructed from the compilation view of designing. The components of the design system include a structural design language, diagnostic compiler, intermediate codes, physical design subsystem, integrated simulators and test pattern generation software.

There is also a design methodology to be used in conjunction with the compiler. It is based on the complete abstraction of physical detail into the system, and partial abstraction of electrical and temporal detail into simple design rules. This is intended to liberate the designer to concentrate on the higher level architectural issues in design where the greatest gains in performance can be made. Taken as a whole the approach provides significant reductions in design time while not sacrificing real estate over more conventional gate array implementations.

2. SILICON COMPILATION

Silicon compilation is not a well-defined term but it can, in its widest sense, be regarded as any translation between a design description and a layout. The earliest example of silicon compilation was probably LAP [Locanthi 78], a set of procedures embedded in a high-level language, in this case SIMULA, for describing IC layouts

0420-0098/82/0000/0377$00.75

e.g. box, polygon. Higher-level constructs such as PLAs could then be programmed using the normal features of a high-level language. This simple idea has now been taken over by many design groups, especially inside universities, and has resulted in many flavours of LAP corresponding to the group's favourite programming language.

The first silicon compiler followed this beginning [Johannsen 78]. It concentrated on deformable cell descriptions so that physical designs could be composed by cell abuttment. Another early contribution to the subject was made by one of the authors [Buchanan 80]. Buchanan's work concentrated on joint physical and structural descriptions of a hierarchical design so that all objects had both coordinate and connectivity attributes. This resulted in a system in which all operations, such as artwork production, stick diagrams, design rule checking and simulation, were performed on the same object. More recently, systems for generating designs in a particular architecture have been demonstrated [Johannsen 81, Rupp 81]. They come closest to true hardware compilers in showing that it is possible to compile a behavioural description into a physical description without exorbitant areal overhead.

These examples show that it is possible to choose a design style and support designing in that style with a particular software environment.

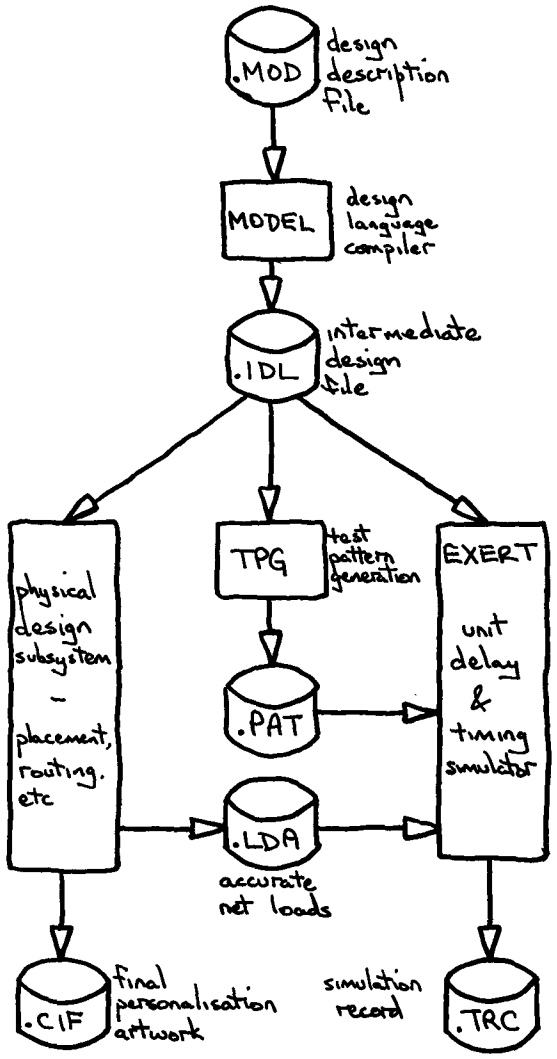

Figure 1: A Gate Array Turn-key Technology

3. DESIGN ENVIRONMENT AND METHODOLOGY

The design environment is strongly based on the idea of programming designs and is shown in Figure 1. There are three primary environments corresponding to the design description, verification, and implementation phases of designing. Within and between environments, functions are modularised to separate programs and communicate via intermediate codes. There is thus no need for central data base support.

Design activity can be thought of as iterating around three conceptual loops in the system, see Figure 2. Structural debugging is via MODEL compilation to yield a design in which there are no topological errors. Functional debugging is performed via a simulator that is built into the system. Both unit delay and timing modes are available and the simulator operates from an intermediate code produced from the MODEL compilation. The intermediate code file contains a list of transistors and their interconnections which is used by the simulator as a basis for a

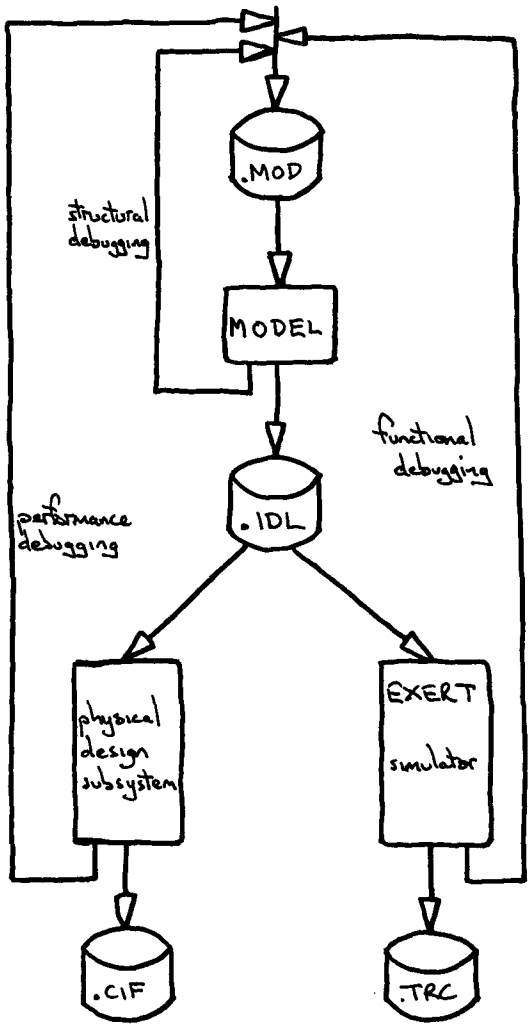

Figure 2 : Design Activity

transistor switch-level simulation. Finally, performance debugging is accomplished via loading analysis and timing simulation.

The notions underlying design in this environment are taken from structured programming. Top down development of a design is carried out by the processes of abstraction and stepwise refinement. It amounts to the identification and fleshing out of major modules at all levels of design abstraction, essentially the development of the module hierarchy. It follows that a key component of a structured design methodology is a notation that allows the natural expression of the partitioning of a design. Additionally, the constructs of the notation provide a mechanism for the "painless" enforcement of good design practice. The MODEL notation has been designed with these principles in mind.

4. DESIGN SPECIFICATION USING MODEL

The fundamental structuring tool in the MODEL language is the part. A part is a unit or module with one or more input signals and one or more output signals. The definition of a part specifies its internal structure in terms of instances of simpler parts and their interconnections. Primitive parts such as NAND and NOR gates, inverters, etc. are predefined in system libraries and do not need to be defined by the user. Higher level libraries e.g. TTL equivalents, are simple to compile. An instance of a part is the use of the part in the definition of a more complex part. Any number of instances of the same part may occur in the same layout.

The part definition specifies its internal form and its interface to the outside world. The potential connections to other parts are defined by arbitrarily-named input and output signals, the formal parameters of the part definition. When an instance of the part is generated, actual signals from the enclosing environment are substituted for the formal signal parameters used in the definition. Local signals may be declared inside a part.

Figure 3 shows a logic diagram for a four-way multiplexor which outputs one of the four input signals depending on which of the four control signals is a 1. A possible specification of the part would be as follows:

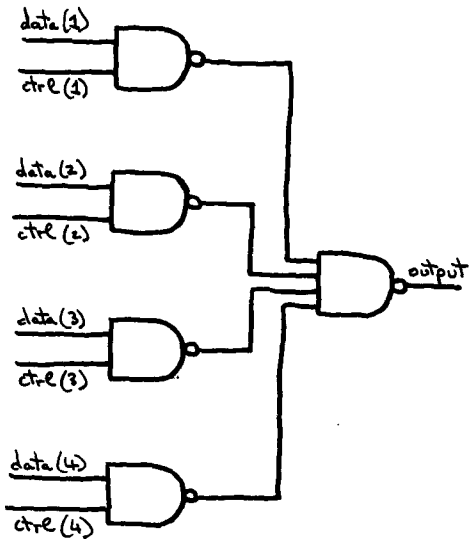

Figure 3 : A 4-way Multiplexor

```
PART mux (data(1:4),ctrl(1:4)) -> output
   SIGNAL temp(1:4)
   nand (data(1),ctrl(1)) -> temp(1)
   nand (data(2),ctrl(2)) -> temp(2)
   nand (data(3),ctrl(3)) -> temp(3)
   nand (data(4),ctrl(4)) -> temp(4)
   nand (temp(1:4)) -> output
END
```

Integer parameters can be declared in the part heading by enclosing the list in square brackets. Local integer variables may be declared inside the part. A way of specifying the same part in a more general form using the numerical parameter feature would be:

```
PART mux [n] (data(1:n),ctrl(1:n)) -> output
   SIGNAL temp(1:n)
   INTEGER i
   FOR i=1:n CYCLE
      nand (data(i),ctrl(i)) -> temp(i)
   REPEAT
   nand (temp(1:n)) -> output
END
```

A further example, the 2 bit decoder shown in Figure 4, illustrates the use of the conditional statement.

```
PART decoder (a,b,latch) -> p(0:3)
   SIGNAL abar,bbar
   INTEGER i
   not (a) -> abar
   not (b) -> bbar
   FOR i=0:3 CYCLE
      nor (IF i&1=0 THEN a ELSE abar ENDIF,
           IF i&2=0 THEN b ELSE bbar ENDIF,
           latch) -> p(i)
   REPEAT
END
```

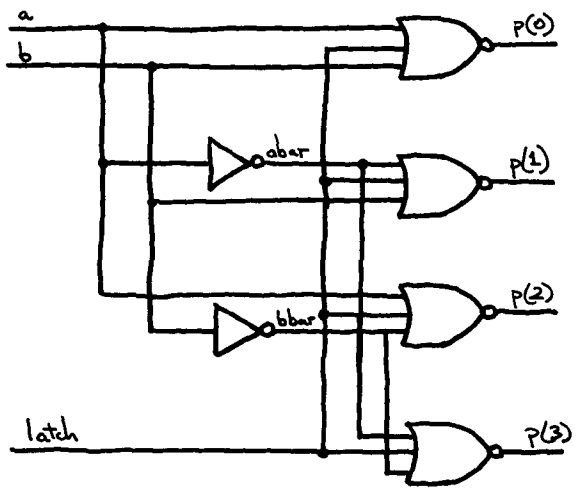

Figure 4 : Two Bit Decoder

A more substantial example, a carry look-ahead adder, is contained in Appendix 1.

5. DESIGN FOR PERFORMANCE, WIRABILITY AND TESTABILITY

In design for performance, because of the fixed device sizes in gate array technologies, it is possible to develop simple rules for estimating best and worst case delays in gating structures. Table 1 shows worst case delays for a particular form of chip image.

	PULLDOWN	PULLUP
Inverter	2	4
Nor	2	4m
Nand	2m	4

m is the number of inputs per gate

Table 1. Worst Case Gate Delay (To be scaled by the technology delay of n-channel device)

These simple rules may be used during the stepwise refinement process to estimate performance. More accurate verification must follow using timing simulation, net loading analysis and circuit level simulation.

There are two approaches to design for wirability: semi-custom gate arrays and full-custom gate arrays. In the first a fixed chip image is preprocessed with a given wire capacity. In all fixed chip images it is possible to generate a design in which wire demand exceeds wire capacity. It is however possible to estimate wire demand given a systems partition. In the case of the MODEL compiler a separate pass over the intermediate code accumulates global and local signal usage to yield a worst case wiring demand. Systems partitions may be adjusted, if possible, to reduce wire demand beneath wire capacity. The alternative approach in full-custom gate array is to generate a chip image with exactly the correct wire space for a particular design. The flexibility of a software approach enables this to be done by fully parameterising the generation of chip images. In a sense this can be thought of as an optimising silicon compiler. In addition to area optimisations it is possible to carry out performance optimisations by adjusting routes to minimise capacitive loading.

It is becoming recognised that design for testability, like performance, is best addressed at the highest levels of design abstraction. If a designer wishes to guarantee testability by avoiding the intractable costs associated with test pattern generation algorithms it is necessary to enforce a register transfer design style and make internal state accessible. Level sensitive scan design is an example of this [Williams 79]. A minimal set of tools to support such a method is made up of automatic test pattern generator for stuck-at faults in combinational logic and a test pattern evaluator for manually generated patterns.

5. SUMMARY

Digital subsystem implementation can be viewed as designing a program that may be compiled to a physical design. It is important that a structured design methodology be used to master the complexity of large designs. A convenient notation, diagnostic programming environment and simple set of design rules, as described in this paper, support this approach. It is conjectured that this silicon compiler will be portable across a range of processes in the same way that high level language compilers can be portable across architectures. Design principles are well understood for the latter and must be discovered for the former to guarantee gate array users independence and flexibility in using products.

REFERENCES

[Buchanan 80] Buchanan I.
"Modelling and Verification of Structured Integrated Circuit Design"
Ph.D. Thesis, Department of Computer Science, University of Edinburgh, 1980.

[Johannsen 79] Johannsen D.
"Bristle Blocks : A Silicon Compiler"
Proceedings of the 16th Design Automation Conference, 1979.

[Johannsen 81] Johannsen D.L.
"Silicon Compilation"
Ph.D. Thesis, Department of Computer Science, California Institute of Technology, 1981.

[Locanthi 78] Locanthi B.
"LAP : A SIMULA Package for IC Layout"
Caltech Display File #1862, 1978.

[Rupp 81] Rupp C.R.
"Components of a Silicon Compiler System"
VLSI 81 Conference Proceedings, Academic Press, 1981

[Williams 79] Willaims, T.W., and K.P.Parker
"Testing Logic Networks and Design for Testability"
Computer, Oct79, pp9-21

APPENDIX 1 : THE CARRY LOOK AHEAD ADDER

The logic diagram for a four-bit carry look ahead adder contains about forty gates and an equivalent number of wires. A similar part can be seen in the TI catalogue, part SN74LS283. However, it is difficult to deduce the function of a design from a network of gates and so it is beneficial for both comprehension and description to structure the design hierarchically as shown in Figures 5,6, and 7. The adder is first subdivided into four bit-slices. Each slice produces its own propagate, generate and sum signals and contains a look-ahead part to produce the carry. The look-ahead part is programmed to produce the appropriate carry for a particular bit-slice. The MODEL encoding for this design is as follows:

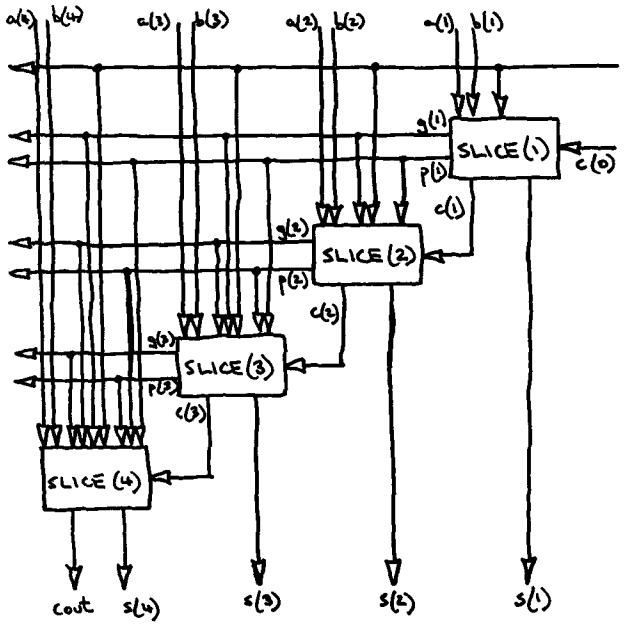

Figure 5 : 4-bit adder

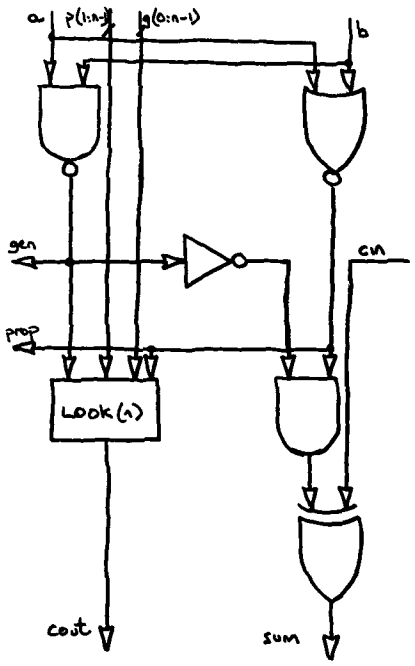

Figure 6: Slice

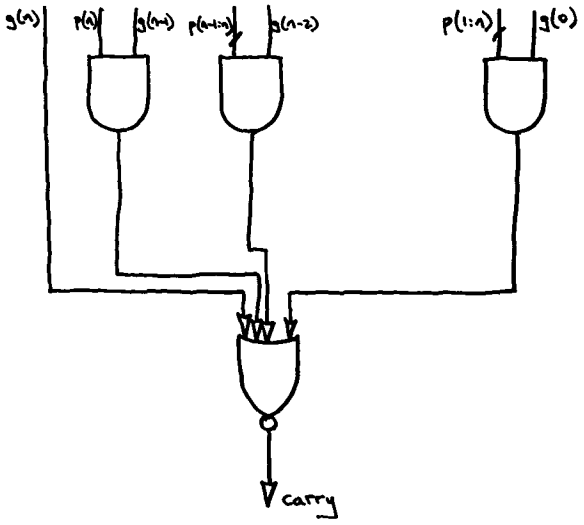

Figure 7: Look

```
{n-bit binary full adder with fast carry}
     {similar to TI part SN74LS283}

Include "library.inc"

Constant bits=4
 Input Pad a(1:bits), b(1:bits), carry in
Output Pad s(1:bits), carry out

Part look [n] (p(1:n),g(0:n)) -> carry
   Signal temp(1:n)
   Integer j
   For j = 1:n  Cycle
      and(p(j:n), g(j-1)) -> temp(j)
   Repeat
   nor(g(n), temp(1:n)) -> carry
End

Part slice[n](a,b,p(1:n-1),g(0:n-1),cin) ->
              sum,cout,prop,gen
   nand(a, b) -> prop
   nor(a, b)  -> gen
   xor(cin, and(prop, not(gen))) -> sum
   look [n] (p(1:n-1), prop, g(0:n-1),gen) -> cout
End

Part adder[n](a(1:n), b(1:n), cin) -> cout, s(1:n)
   Integer j
   Signal c(0:n), p(1:n), g(0:n)
   not(cin)  -> g(0)
   not(g(0)) -> c(0)
   c(n) -> cout
   For j = 1:n Cycle
      slice [j] (a(j), b(j), p(1:j-1),
                 g(0:j-1), c(j-1))        ->
                s(j), c(j), p(j), g(j)
   Repeat
End

adder[bits](a(1:bits),b(1:bits),carry in) ->
           carry out, s(1:bits)

Endoffile
```