PROGRAMMING LANGUAGES FOR HARDWARE DESCRIPTION

Peter Robinson and Jeremy Dion

Cambridge University Computer Laboratory
Corn Exchange Street
Cambridge CB2 3QG England

Abstract

This paper describes work done as part of a design automation system for gate arrays. The system provides no hardware description language and circuits are described in the same language in which the system is implemented, Modula-2. We claim that modern programming languages are sufficiently general to be good hardware description languages, and that a design automation system which is based on this principle will be simpler and more extensible than one based on a specialized hardware description language.

Introduction

A design automation system for gate arrays may be defined as a program which manipulates four representations of a circuit:

(a) Functional - often present only in the designer's mind but ideally formally specified and verified.
(b) Structural - the decomposition of the circuit as an interconnected network of simpler circuit elements, ultimately to the gate level.
(c) Physical - a set of mask definitions usable for fabrication.
(d) Actual - the final packaged chip for which tests must be generated by the design automation system.

The automation task consists of two main parts: construction, the conversion of a functional or structural description into a physical description, and verification that all four representations of the circuit are equivalent.

Ideally, the circuit designer should be able to specify a circuit in functional terms and have it translated automatically into a structural form which is then laid out automatically. This is already possible for restricted types of circuit [Siskind, 1982], but for general sequential circuits it cannot yet be done with acceptable efficiency. The design automation system we have built therefore requires the designer to specify the structure of the chip as a hierarchical decomposition of circuit elements in which the leaves are gates, but also allows a functional description of circuit elements to be given. The functional and structural descriptons may then be compared by simulation, and a physical description of the mask derived by automatic layout. A general view of the system is given elsewhere [Robinson, 1982].

Structural and functional descriptions have, of course, been in use for many years. That used by Texas Instruments [Hightower, 1981] shares our approach to the style of description, but uses a specialised language rather than a general programming language.

Structural description

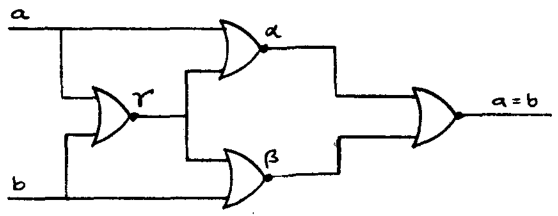
In the form of circuit description advocated here, the designer defines a circuit using a standard, general-purpose programming language, and compiles it with a standard compiler. No extensions to the language are required, and the design automation system appears to the designer (or programmer) as a collection of procedures in standard libraries. When the program describing the circuit is run, it builds a data structure which describes the elements of the circuit and their interconnection. This approach of compiling and running a circuit description program has the same effect as interpreting a description in a conventional hardware description language (HDL); both produce a data structure which is then manipulated by the design automation system.

The language used is Modula-2 [Wirth, 1980], a modern programming language derived from Pascal with a modular structure similar to Ada [DOD, 1980] or Mesa [Mitchell, 1979]. The language has the obvious conditional, loop and procedure structures. These features are as useful for describing hardware as for describing programs - loops for generating repeated structures, tests for building structures depending on the environment, and procedures for defining blocks of

0738-100X/83/0000/0012$1.00 © 1983 IEEE

related logic - and are often present in various forms in specialized HDLs. Modula-2 is also strongly typed and has excellent facilities for separate compilation which are as helpful to the hardware as to the software designer. Type checking eliminates many errors at compile time, and good separate compilation allows the construction of program libraries of useful circuit elements in a general and extensible way.

Figure 1 shows a fragment of a Modula-2 circuit description and the equivalence circuit it builds when executed.



```
PROCEDURE Equivalence
  (name: ARRAY OF CHAR; a, b: SIGNAL): SIGNAL;
VAR
  alpha, beta, gamma,
  equivalence: SIGNAL;
BEGIN
  BeginNode ("equivalence", name);
  gamma := Nor2 ("gamma", a, b);
  alpha := Nor2 ("alpha", a, gamma);
  beta := Nor2 ("beta", gamma, b);
  equivalence := Nor2 (name, alpha, beta);
  OuputPin (0, equivalence);
  InputPin (1, a);
  InputPin (2, b);
  Model (EquivalenceModel, NIL);
  EndNode (name);
  RETURN equivalence;
END Equivalence;
```

Figure 1: Structural specification of an equivalence circuit.

The procedure takes three arguments; a string, which will distinguish this particular equivalence circuit from all others, and two input signals a and b which represent signals (wires) in the circuit. It returns a signal which holds the result of computing a = b. The procedure is a template for constructing an equivalence circuit; a circuit which contained 50 such circuits would be made by calling Equivalence 50 times, not by writing 50 times as much text.

All the circuit elements described in this way are implicitly concurrent - the sequential execution of the circuit description program builds a data structure whose elements are all active simultaneously, and no explicit specification of concurrency is necessary.

The main purpose of the procedure is to create a node in the circuit representation, and it does this by making a sequence of calls to other procedures in the design automation system. BeginNode, in the first line, starts the description of a new node in the circuit; it is defined to be of type "equivalence", with name name. The effect of BeginNode is the creation of a 'blackbox' in the circuit. It must now be defined in terms of simpler circuit elements and connected to the rest of the circuit. First some internal signals alpha , beta and gamma are defined by calling the Nor2 procedure. This procedure makes both a NOR gate and the signal which it drives, and then connects the NOR gate to its two input signals, in the same style of operation as Equivalence itself. The signal equivalence which will be returned by the procedure, is defined in the same way.

The next four lines are only needed if the equivalence circuit is to be simulated directly (this will be discussed further in the next section). The calls of InputPin and OutputPin attach signals to numbered 'pins' of the equivalence circuit. The call of Model tells the simulator that the procedure EquivalenceModel is a behavioural model for an equivalence circuit which should be used in simulation. It also gives the simulator a state vector which is to be passed to EquivalenceModel on each call. Since an equivalence model is purely combinational, no state is needed and the conventional value NIL is given.

The call of EndNode is made to terminate the description of this node in the circuit. Its argument is the same 'name' which was used in the call of BeginNode , and is used to check that each BeginNode has a matching EndNode. In the last line, the signal which represents the wire on which a = b is computed is returned to the caller.

The structural description generated in this way is hierarchical. At each level, a number of nodes are defined and connected together, and each of these nodes is internally further subdivided into simpler elements. The Equivalence procedure builds a node which contains four sub-nodes and some internal wiring to connect them. These sub-nodes are all simple NOR gates and so happen to be leaves of the tree, but the Equivalence procedure would not have been written differently if they had not.

Alternative forms of description

We have found that designers find this scheme of circuit desciption very attractive, and enjoy the precision afforded by written text. However, there are two major deficiencies: the description is somewhat verbose and it lacks the intuitive feel of a circuit diagram. We believe that both these problems can be solved quite simply by the use of pre-processors.

The verbosity stems directly from the use of a programming language. When a signal is defined, it must be both declared as a variable in the program and given a name to be attached to it in the circuit model, resulting in the same name being typed twice. In a special purpose language, a pun could be made between the name of the variable and the name of the signal, but the actual names of variables are lost by the time our program is run and only the quoted strings survive. Similarly, the calls to Begin Node and End Node and the various pin descriptions could be derived from the structure of the procedures and arguments in the source text.

The solution to this problem is clear. Given a language capable of describing hardware, a program could be written to translate it automatically into a Modula-2 description, and this translator could make the necessary duplications of text. Indeed, the principle could be extended so that different hardware description languages, suited to the styles of different designers, could all be translated into the common Modula-2 description. We have such a pre-processor that takes a macro description language for circuits loosely based on that used by the UK 5000 project [Grierson, 1983] and translates it into Modula-2.

Circuit diagrams are somewhat more tricky. There seem to be two different uses made of circuit diagrams by designers: during the early stages of design, the picture helps them in the analysis of the modular structure of the circuit, and during the more detailed stages of design, the picture illustrates precise signal paths through a maze of gates. However, both these uses can be seen as allowing graphical presentation of different levels in the hierarchical description.

In this sense, circuit diagrams (and block diagrams) are just another form of structural description and programs could be written to translate between pictures and program descriptions, and to check that a distinct circuit diagram and program description describe the same underlying circuit. These translations are no more difficult than the expansion described above, except that they are likely to involve the use of interactive graphics, and this always requires care to ensure that the ergonomics of the system are satisfactory.

Functional description

A functional description defines the behaviour of a node in the circuit description tree as a finite state machine with timing information. The procedure which models an equivalence circuit is given in Figure 2.

```
PROCEDURE EquivalenceModel
    (pin: CARDINAL; a: ADDRESS);
BEGIN
  IF Pin (1) = Pin (2) THEN
     ChangePin (0, One, 8, 10);
  ELSE ChangePin (0, Zero, 8, 10);
  END;
END EquivalenceModel;
```

Figure 2: Functional specification of an equivalence circuit.

This procedure is called by the simulator whenever one of the input signals of the equivalence circuit changes. The first argument is the number of the pin which has just changed, and corresponds to the Input Pin numbering used in the structural definition. The second argument is a state vector which the simulator passes to the procedure on each call. As previously mentioned, an equivalence circuit is purely combinational, so it needs no state information. For a register or memory or processor model, however, the state vector would be consulted to determine how to react to each input change. In this case the behaviour is particularly simple; on each input change, the model computes the equality of the logic levels on its two inputs in the expression Pin(1) = Pin(2), and changes its single output accordingly. ChangePin defines the pin which is to change (0), the new value which it is to take (logic Zero or One) and the earliest and latest times at which the change can occur (between 8 and 10 nanoseconds from the current simulation time).

The equivalence circuit as defined above could be simulated either at the functional level or at the NOR gate level. In the current implementation of the simulator, however, the designer may specify either a structural description for a circuit element in terms of sub-components, or a functional description of the element as a procedure, but not both. Work on removing this restriction is in hand.

For an example where state information is used, consider the counter shown in Figure 3. A distinct STATE record is created for each instance of the counter, to hold the width of the output bus and the current value. Whenever a falling edge is detected on the clock input, the current value is asserted on the output bus after a delay of between 40 and 50 ns, and the value is incremented. A more elaborate model could detect too rapid a sequence of changes on the

```
TYPE STATE = POINTER TO RECORD
    Width,
    Value: CARDINAL
  END;

PROCEDURE Counter (name: ARRAY OF CHAR;
  clock: SIGNAL; output: ARRAY OF SIGNAL;
  initial: CARDINAL);
VAR
  state: STATE;
  i: CARDINAL;
BEGIN
  ...
  Input Pin (0, clock);
  FOR i := 0 TO HIGH (output) DO
    Output Pin (i+1, output [i]);
  END;
  NEW (state);
  WITH state^ DO
    Width := HIGH (output);
    Value := initial;
  END;
  Model ( CounterModel, state);
  ...
END Counter;

PROCEDURE CounterModel
  (pin: CARDINAL; state: STATE);
BEGIN
  IF PinValue (0) = Zero THEN
    WITH state^ DO
      ChangePins (1, Width+1, Value, 40, 50);
      INC (Value);
    END;
  END;
END CounterModel;
```

Figure 3: Functional description of a counter.

clock and signal the fault in some convenient way - either by letting the output bus go undefined or by writing an error message.

Precisely the same mechanism is used to describe the circuits external to the chip such as processors or memory. This gives the designer a kit of procedures modelling the components of the system, which can be assembled to form test environments. Such a scheme replaces the conventional presentation of sequences of test patterns to the pins of a chip being simulated, and is substantially more flexible.

Discussion

The circuit description method outlined above is both simple and general. The NOR gate, to which all circuits are ultimately reduced, is described by two procedures which are very similar to Equivalence and EquivalenceModel ; a NOR gate is just a node with a modelling procedure and no sub-nodes. Because circuits are defined in the same language in which the system is implemented, any designer can add new primitive elements of his own design, and can build up libraries of useful circuit components in a completely general way. The resulting design automation system is particularly flexible, and can be extended in ways not considered by the original designers. A prototype version of the simulator, originally designed for use with a NOR logic gate array, was quickly converted for simulation of a NAND logic array with very different delay characteristics.

We claim that a good HDL needs much of the mechanism of a general-purpose programming language. Language design is itself a difficult problem, as the current controversy over Ada amply illustrates, and building a reliable and efficient translator for a well-defined HDL is a serious effort. It is not clear that the syntactic convenience of a specialized HDL justifies this effort and the resulting loss of flexibility.

The use of a programming language obviates the need to learn distinct languages for structural and functional description. If a new language has to be learned, it seems reasonable to use the same high-level programming language as is used throughout the system, a language that has been designed with an eye to clarity of expression and ease of correct programming. Moreover, once Modula-2 has been learned, the designer is free to experiment with algorithms in the DA system itself (there is a controlled way of doing this without disturbing other users!). This is actually a requirement when specifying sophisticated functional models anyway, but offers the possibility of simulating unusual effects, such as monostables, which can be built using the components on an uncommitted logic array, and the designer can even program alternative layout strategies to be invoked when the automatic system fails.

If a programming language has to be learned, we would suggest that Modula-2 is as good a choice as any, and better than most. Prototypes for the layout and verification parts of this system were programmed in BCPL and Algol-68C, and these languages were accordingly used as hardware description languages. We have found Modula-2 to be a better programming language, and a better HDL, than either of these.

There are three features that particularly serve to illustrate this. The language has proper facilities for the definition of abstract data types and for checking that these types are respected - this is a boon to the programmer, inhibiting numerous careless mistakes. Proper facilities for separate compilation are provided, encouraging a modular style of programming, which in turn leads to a well structured program and generally encourages correctness. This extends into a general library mechanism which can be used to present families of useful modules in the DA system; for example, the facilities of standard SSI TTL chips can be presented in such a library. Moreover, because there is no distinction between standard and specialised parts of the system, users can offer their designs as libraries for other users in a completely general way.

Whilst the language is conducive to the writing of correct programs, the correct description of correct circuits is a different problem. The other parts of our DA system incorporate a number of verification stages to assist with this.

The problems with the system are its verbosity and the difficulty some logic designers have in adapting to the written word as the master description of a circuit. These have been discussed above, and the experience of several users of our system has been that they come to appreciate the precision, power and flexibility of the Modula-2 system.

Acknowledgements

The research work described in this paper has been performed under a research contract from the Science and Engineering Research Council's Rutherford Appleton Laboratory. Many people at the Cambridge University Computer Laboratory have contributed ideas to this work; Andy Hopper, Roger Needham, David Wheeler, Maurice Wilkes and Neil Wiseman deserve special mention. Mick Jordan provided a program development environment for Modula-2 that accelerated the progress of the work immeasurably.

References

DOD, 1980:
Reference Manual for the Ada Programming Language; United States Department of Defense, July 1980.

Grierson, 1983:
UK 5000; John Grierson, British Telecom, with B Cosgrove, R Daniel, RE Halliwell, IH Kirk, JA Mc Lean, JM Mc Grail and CO Newton; 20th ACM/IEEE DA Conference, June 1983.

Hightower, 1981:
Automated logic arrays and the customer interface; D Hightower and M Roberts, Texas Instruments; Proceedings VLSI Compcon, Spring 1981.

Mitchell, 1979:
Mesa Language Manual Version 5.0; J Mitchell, W Maybury and R Sweet, Xerox Palo Alto Research Centre.

Robinson, 1982:
Design aids for uncommitted logic arrays; Peter Robinson and Jeremy Dion, Cambridge University Computer Laboratory; 2nd International Conference on Semi-custom ICs, November 1982.

Siskind, 1982:
Generating custom high performance VLSI designs from succinct algorithmic descriptions; JM Siskind, JR Southard and KW Crouch, MIT Lincoln Laboratory; January 1982.

Wirth, 1980:
Modula-2; N Wirth, ETH Zurich; March 1980.