A BDD ENGINE FOR LOGIC VERIFICATION

Dariusz Caban*

Logic verification is based on the comparison of a circuit behavioural specification with its structural realisation. It plays an important part in VLSI design. The circuit descriptions are expressed in a hardware description language. A modified language may simplify the verification process, e.g. the NODEN HDL. Logic verification is based on the transformation of different levels of circuit description to a common, canonical form. Binary Decision Diagrams (BDD) are used for this purpose because of their capability to produce simple models for the results of arithmetic addition. Logic verification is a numerically complex task, which limits its usefulness. Parallel processing on a multi-transputer system can make it more attractive. It is proposed to introduce the parallel processing at the level of constructing the decision diagrams (BDD Engine).

1. Introduction

Logic verification is a formal method of establishing the correctness of digital circuits. It compares the expected and extracted descriptions of a circuit. Unlike logic testing verification does not depend on any data stimuli. Thus it can authenticate a design with total assurance, subject only to errors in the circuit specification (Hachtel. and Jacoby, 1988).

The design process of a digital circuit is split into a number of consecutively executed stages, with numerous loop-backs. These stages are roughly as follows:

Specification. A formal model of the required behaviour is developed. This is expressed in a hardware description language (VHDL is the norm at present).

Functional testing. This is a means, along with syntax checks, to determine the correctness and completeness of a specification. At this level formal verification is not feasible as there is no model of correct behaviour.

Logic design consists of a number of distinct stages, in which the circuit description is transformed and substantiated to produce a technology dependent design. The result is a network of cells (gates, registers, etc.). Logic designs are traditionally expressed by graphical schematics. HDL descriptions are also used.

Implementation. Mainly the decomposition, placement and routing or equivalent processes. Implementation does not change the logic description, but creates the data needed to produce the device (printed board, field programmable device, semi-custom ASIC chip or a full-custom VLSI product). The realisation may affect the logical

* Department of Electrical and Electronic Engineering, University of Bristol, University Walk, Bristol BS8 1TR, UK (on leave from Wroclaw Technical University)

description (timing is a typical example), so the design description is back-annotated with this information.

Testing, logic verification and timing analysis. These are aimed at uncovering any design errors before manufacturing the device (Gupta A., 1992). This is crucial in case ASIC designs, where the cost of an error is very high. Formal verification is always preferable if the design complexity does not preclude it. In fact, design verification is often done at every step of circuit development and is not restricted to the last stage of the design.

Implementation verification is aimed at authentication of the physical implementation. Unlike logic verification, this is based on checking if the implementation conforms to (geometrical) constraints. Implementation process is often fully automated or computer-aided. This ensures implementation correctness without further verification.

2. Logic Verification

It is best to describe logic verification by the involved stages of processing (Pygott, 1988). The two models of the designed circuit, obtained from specification and from the implemented cell network, form the input to logic verification process. These models are expressed in a hardware description language.

The first stage of processing is based on description compilation. An intermediate form is produced as a result. This form is more compact and particularly suitable for further processing. The important task at this stage is the syntax analysis which ensures that the descriptions do not contain improperly formed constructs.

In the next stage (analysis) the intermediate form is executed. This results in the construction of a set of logic functions. These functions are expressed in a canonical form which ensures that equivalent circuit descriptions will produce exactly the same set of functions. The choice of the canonical form is crucial to logic verification as it determines the complexity of the description.

The last stage is logic comparison. The previous stages are performed separately for the specification and implementation. Comparison can be done by textual testing of the two canonical descriptions character by character (textual comparison). This is the fastest method but it can be used only to verify equivalence of circuit descriptions. Usually, there are ambiguities in the specification which are resolved in the design process. In such cases the aim of verification is to test if the implementation falls within the bounds set by the specification. This can be done by constructing the Boolean function of implication and testing if it is always TRUE (comparison by logic implication). This is very similar to the tasks performed in the analysis stage, though some simplifications are possible.

The logic circuits are traditionally classified as either combinational or sequential, synchronous and asynchronous. The approach to logic verification is essentially general but each class of circuits introduces specific problems. Combinational circuits are straightforward to verify on the basis of logic function transformations.

Sequential circuits need to be described by their set of permissible states and by the state--transition functions. If the specification and realisation state sets are similar, verification can be achieved by comparing the state-transition function. Automatic identification of corresponding states of state machines is not a trivial problem, though. It can be avoided by comparing the results of logic simulation of the two machines.

Asynchronous circuits introduce the problem of continuous time in the functional descriptions. This cannot be handled by the approach discussed in my presentation. Methods based on temporal logic can be used as shown in (Gupta, 1992).

3. NODEN Hardware Description Language

Hardware verification poses the requirement that circuit specification and realisation are expressed in a formal language. General hardware description languages, such as VHDL, ELLA or VERILOG, can be used. The expressive capability of these languages is sufficient use them both for specification and implementation. This is important as it allows the use of the same tools for description compilation and analysis.

The mentioned HDL languages do not have any mechanisms by which one could simplify the verification process. Logic verification is numerically very complex and at present it can be applied only to relatively simple circuits. The designer can introduce extra information which significantly reduces the complexity of the task, as is demonstrated further in the text. This is the motivation for developing specific languages for verification.

The NODEN Hardware Description Language has evolved from ELLA as a specific language for logic verification (Pygott, 1988). Its capability is limited to synchronous circuits only. The language has been tested in practice on the VIPER project. It introduces a number of mechanisms by which a designer can influence the verification process. These are discussed in the following paragraphs.

Logic verification can only be applied to descriptions of a certain size. This is presently much smaller than the complexity of a VLSI device. It can still be used to verify fragments of the design. To this end, NODEN introduces two kinds of hierarchical components of a description: circuits and blocks. Verification is discontinued at the highest description of "block" type. Circuits are verified in a simplified way:

- by verifying all the blocks in their description,

- by comparing that all blocks in the specification correspond to blocks in the implementation,

- by comparing that all the blocks are similarly connected in the circuit description of specification and implementation.

Thus, verification of much larger circuits is possible provided they are similarly structured at specification and implementation.

Another problem is caused by the correspondence between the specification and implementation inputs/outputs . There is no valid reason to use the same names or data types in the compared descriptions. Thus, verification should be performed for every

```
TYPE int4 = INT[0..15],
     int5 = INT[0..30].
MAP UNCNST = int4 -> _integer.
MAP CONSTR = _integer -> int5.
BLOCK ADDER = (int4: a b) -> (^int5: c):
     CONSTR(UNCNST a + UNCNST b)).
```

Fig. 1. Specification of a 4-bit adder.

mapping between the inputs/outputs of corresponding blocks. Since the canonical representations depend on the order of variables, this means repetition of the analysis and comparison stages for every order of input variables. NODEN avoids this by requiring the designer to annotate the implementation level description with "associate bodies". These are special constructs which map block names and the names and types of inputs/outputs to the names used in specification.

NODEN adopts the associate bodies to the analysis of sequential circuits, too. As already mentioned, the problem of analysing these circuits lies in the identification of corresponding states. NODEN requires the designer to supply this mapping in the associate bodies.

The use of NODEN at specification and implementation level is best illustrated by example. Figure 1 shows the specification-level description of a 4-bit adder. Figure 2 represents the implementation of the same adder using cells of functionality: XOR2, XOR3 (2- and 3-input exclusive or) and CR (1 bit carry). Figure 3 shows the associate body which must be attached to the implementation.

```
DELAY DL = bool.
FN XOR2(bool: a b)    -> bool: (a AND NOT b) OR (NOT a AND b).
FN XOR3(bool: a b c) -> bool: a XOR2 b XOR2 c.
FN CR(bool: a b c) -> bool: (NOT c AND a AND b) OR (c AND (a OR b)).
FN ADD_IMP([4]bool: a b) -> [5]bool:
BEGIN
MAKE bool c2 c3 c4.
MAKE DL: r1 r2 r3 r4 r5.
LET c1 = a[0] AND b[0].
JOIN (a[1] XOR2 b[1]) -> r1.
JOIN CR(a[2], b[2], c1) -> c2.
JOIN XOR3(a[2], b[2], c1) -> r2.
JOIN CR(a[3], b[3], c2) -> c3.
JOIN XOR3(a[3], b[3], c2) -> r3.
JOIN CR(a[4], b[4], c3) -> r5.
JOIN XOR3(a[4], b[4], c3) -> r4.
OUTPUT  (r5 r4 r3 r2 r1)
END.
```

Fig. 2. Implementation of a 4-bit adder.

There is a number of reasons for using the NODEN language and verification system in the reported project. The motivation for choice is outlined in the following ideas:

- a readily available verification specific language,

- documented analyser interface , which allowed concentration of efforts on construction and manipulation of logic representations (instead of language compilation).

From the point of view of practical applications, it would have been preferable to use a language derived from VHDL , but such was not available.

4. Introduction To Binary Decision Diagrams

Binary Decision Diagram is a graphic representation of Boolean functions. This representation was introduced in (Akers, 1978). Formal definition of the Diagrams can be found there and in all the other cited papers on BDDs. An informal description is given hereafter:

- A BDD is a directed acyclic graph, in which every node represents a Boolean function.
- There are two sink nodes corresponding to constant functions TRUE and FALSE.
- Every non-sink node is attributed with a logic variable. Two output edges are drawn from each node. They are labelled as 0 and 1 edge.
- The Boolean function represented by a node is determined recursively on the basis of functions represented by the two nodes pointed to by the edges. The Shannon decomposition is used:

$$\overline{v} \cdot f_{\overline{v}} + v \cdot f_v \qquad (1)$$

  where $v, \overline{v}$ are the variable attributed to the node and its complement, and $f_v, f_{\overline{v}}$ - the functions represented by the two nodes pointed by the edges, labelled 1 and 0 respectively.

Ordered BDD is obtained by imposing extra constraints on the construction of a diagram. The input variables are ordered and the diagram so structured that every source to sink path passes the variables in ascending order.

Reduced Ordered Binary Decision Diagram (ROBDD) imposes a further constraint that the diagram must be reduced, i.e. each node must represent a unique function. In effect if the decomposition of two different functions yields the same subfunction, both will point to the same node in the BDD. This type of diagrams was proposed by (Bryant, 1986). It is a canonical representation of a Boolean function that can be used in verification.

Typed Reduced Ordered Binary Decision Diagram (Madre and Billon, 1988) is obtained by introducing additional edge attributes. The most commonly used is the edge complement operation (marked with a dot at the end of an edge). This modification can

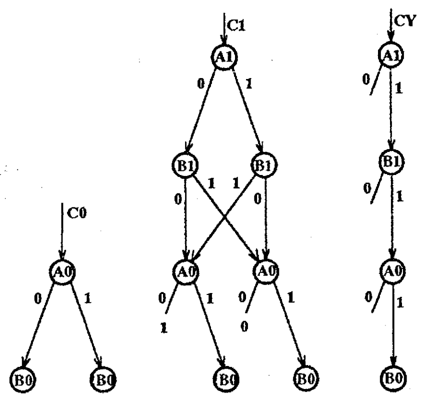

Fig. 4. BDD representation of a two-bit adder.

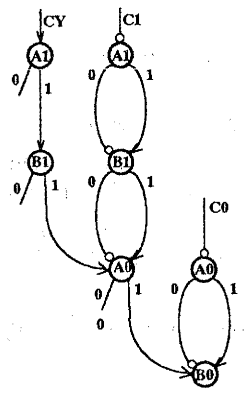

Fig. 5. A Shared Typed BDD representing the same adder.

significantly reduce the complexity of some diagrams. Furthermore, it simplifies the BDD construction algorithm. Unfortunately, the typed diagrams are not canonical without supplementary constraints, as follows from the de Morgan equations. Uniqueness of representation is achieved by imposing the rule that 1-labelled edges cannot be complemented.

Each node of a BDD represents a logic function. Only the root node represents a function of all the input variables in an ordered diagram. Multi-rooted diagrams are introduced to represent multiple functions. Such diagrams are called **Shared Binary Decision Diagrams.**

The use of BDDs for Boolean function representation is illustrated in Figure 4, which shows the graphs of functions of a two-bit adder:

$$\begin{aligned} C0 &= A0 \oplus B0 \\ C1 &= A1 \oplus B1 \oplus (A0 \& B0) \\ CY &= A1 \& B1 \& A0 \& B0 \end{aligned} \tag{2}$$

Figure 5 shows the same functions represented by a Shared Typed BDD.

5. The Parallel BDD Engine

Logic verification can be speeded up by parallel processing. This is achieved by implementing a BDD Engine hosted on an array of transputers. The transputers are connected by their links into a toroidal configuration as shown in Figure 6. The input block description is initially analysed in the ROOT transputer, which is the only one capable of communicating with the outside world.

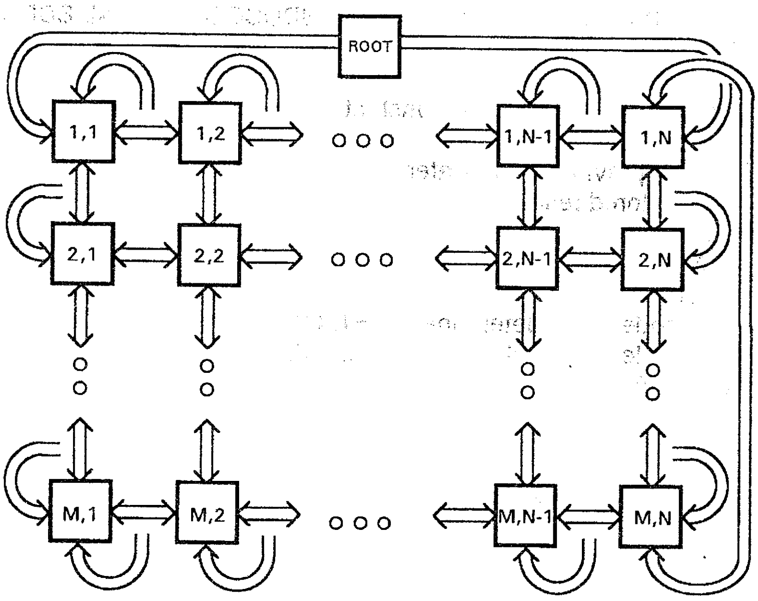

Fig. 6. Hardware configuration of the BDD Engine.

The fundamental algorithm used for BDD construction is based on (Brace, Rudell and Bryant, 1990). It is represented in the following pseudo-OCCAM notation (Fig. 7). The procedure MLApply is strongly recursive - each diagram is constructed by the application of MLApply to its two subdiagrams. Parallelism is achieved by replacing the recursion in this algorithm with requests for "child" MLApply processes. Thus the parent process, blocked until the subdiagrams are constructed, invokes two child processes which run completely independently.

In fact the MLApply process does not generate any processes directly. It only sends messages, requesting subdiagram processing. These messages are transmitted to different processing nodes (transputers), where they are queued. The requests are dealt with one by one. A new message is processed only after the previous one is finished or blocked. This is preferred to unproductive switching between processes simultaneously evaluating diagrams on the same transputer.

The transputer toroid is a distributed processing and distributed data system. Thus the BDD coding tables and the evaluated and stored construction results are distributed between the transputers. The BDD Engine was constructed so that a request for BDD construction is always routed to the same transputer node, so the results of previous calculations are available locally. This is important - benchmarks indicate that as much as 50 % of the requests are repetitions of previous ones.

The distribution of BDD coding tables necessitates some other types of messages, i.e. requests for BDD encoding and decoding, which can be realised only in the node containing the proper coding table. Some other messages are introduced for diagnostics and control.

```
BDDCODE FUNCTION MLApply_OP(VAL BDDCODE Z1, VAL BDDCODE Z2)
 VALOF
  IF
   Z1 or Z2 represents a logic constant
    result:=Z1 OP Z2
   Z1 OP Z2 previously evaluated
    result:=stored result
   TRUE
    SEQ
     PAR
      Decode Z1  -- determine v1, F1, G1
      Decode Z2  -- determine v2, F2, G2
     v:=min(v1,v2)
     PAR
      IF
       v1 <> v
        F1:=G1:=Z1
       v2 <> v
        F2:=G2:=Z2
       TRUE
        SKIP
     PAR
      r1:=MLApply_OP(F1,F2)  -- recursion
      r2:=MLApply_OP(G1,G2)-- recursion
     sgn:=normalize r1,r2 pair to conform to rules of uniqueness
     result:=search for entry (v,r1,r2) or add new entry to coding table
     result:=result*sgn
     store OP result for future reference
 RESULT result
:
```

Fig. 7. Fundamental BDD construction algorithm.

The messages are handled at each node by a supervisor process which invokes one of six message handling processes. The MLApply algorithm is split into a number of these processes in such a way that none can be blocked before completion. If the MLApply processing needs to wait for information from another transputer node (i.e. for the reply to a request for subdiagram construction, BDD decoding or BDD encoding), this occurs at a point between the node processes. The processes are so constructed that no two can access the same data. These processes are as follows:

Process 1. Searches if the requested BDD has been previously constructed or is currently being constructed. If so, it invoke Process 4. Otherwise, it schedules decoding of argument BDDs.

Process 2. Finds if the result can be evaluated directly. If so, it passes the result to Process 4. Otherwise, it requests the construction of subdiagrams.

Process 3. Schedules the encoding of the diagram.

Process 4. Sends the result to the requesting node.

Process 5 (DECODE). Searches the local BDD coding table to determine the node variable and subdiagrams and sends the result to the requesting node.

Process 6 (ENCODE). Searches the local coding table for the presence of specified BDD. It creates a new position if one is non-existent. Finally, it sends the encoded BDD identifier to the requesting node (this identifier consists of the node number and position in the coding table).

Acknowledgements

The presented research is funded at the University of Bristol by a SERC grant.

I wish to thank Clive H. Pygott from RSRE, who let me have the NODEN interface documentation and helped me develop the concept of parallel BDD Engine.

References

Akers S.B. (1978): *Binary Decision Diagrams.-* IEEE Trans. Computers, v.27, No.6, pp.509-516.

Brace K.S., Rudell R.L. and Bryant R.E. (1990): *Efficient implemenation of a BDD package.-* Proc. 27th ACM/IEEE DAC, pp.40-45.

Bryant R.E. (1986): *Graph-based algorithms for Boolean function manipulation..-* IEEE Trans. Computers, v.35, No.8, pp. 677-691.

Hachtel G.D. and Jacoby R.M. (1988): *Verification algorithms for VLSI synthesis.-* IEEE Trans. Computer-Aided Design, v.7, No.5, pp. 616-640.

Gupta A. (1992): *Formal hardware verification methods: a survey.-*Formal Methods in System Design,v.1, No.2/3, pp.151-238.

Madre J.C. and Billon J.P. (1988): *Proving circuit correctness using formal comparison between expected and extracted behaviour.-* Proc. 25th ACM/IEEE DAC, pp.205-210.

Pygott C.H. (1989): *The NODEN Hardware Description Language.-* RSRE Report No. 89011, Ministry of Defence.

Pygott C.H. (1989): *The Algebra of the NODEN Analyser.-* RSRE Report No. 89012, Ministry of Defence.