Technical Report

UCAM-CL-TR-109
ISSN 1476-2986

Number 109

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Hardware verification of VLSI regular structures

Jeffrey Joyce

July 1987

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Hardware Verification of VLSI Regular Structures

Jeffrey Joyce
Computer Laboratory
University of Cambridge

Introduction

Many examples of hardware verification focus on hierarchical specification as a means of controlling structural complexity in a design. This method hides internal details of the implementation at each stage in the hierarchy. Iteration is another method used to control structural complexity. For example, [Camilleri86] describes the formal specification and verification of an n-bit adder using a primitive recursive definition to specify the iteration of adder stages. This paper presents a third method of controlling structural complexity in hardware specifications, namely, the mapping of irregular combinational functions to regular structures such as ROMs and PLAs.

Regular structures often result in solutions which are economical in terms of area and design time. The automatic generation of a regular structure such as a ROM or PLA from a functional specification usually accommodates minor changes to the functional specification, for example, a last-minute change in the microcode of a microprocessor, without any changes to the rest of the design. In addition to these well-known advantages, we suggest that the use of regular structures is an advantage when proving the correctness of hardware designs.

The mapping of irregular combinational functions to a regular structure separates function from circuit design. This paper shows how this separation can be exploited to formally derive a behavioural specification of a regular structure parameterized by the functional specification. Furthermore, this separation of function from circuit design shifts the focus of the verification task from the circuit design to the algorithm which maps the functional specification to an instance of the regular structure.

To illustrate these ideas, we describe the formal specification and verification of a ROM implemented by two NOR arrays. One of the NOR arrays implements an n-bit decoder while the other NOR array stores the contents of the ROM. In section 2 of this paper we informally describe the construction of a NOR array from MOS level devices. Section 3 briefly describes the formal specification of hardware in higher-order logic. This is followed by a simple model of MOS level behaviour in Section 4. This low-level model is then related to boolean logic by abstraction functions defined in Section 5. Section 6 outlines the formal specification of a NOR array parameterized by a "placement function" which determines the placement

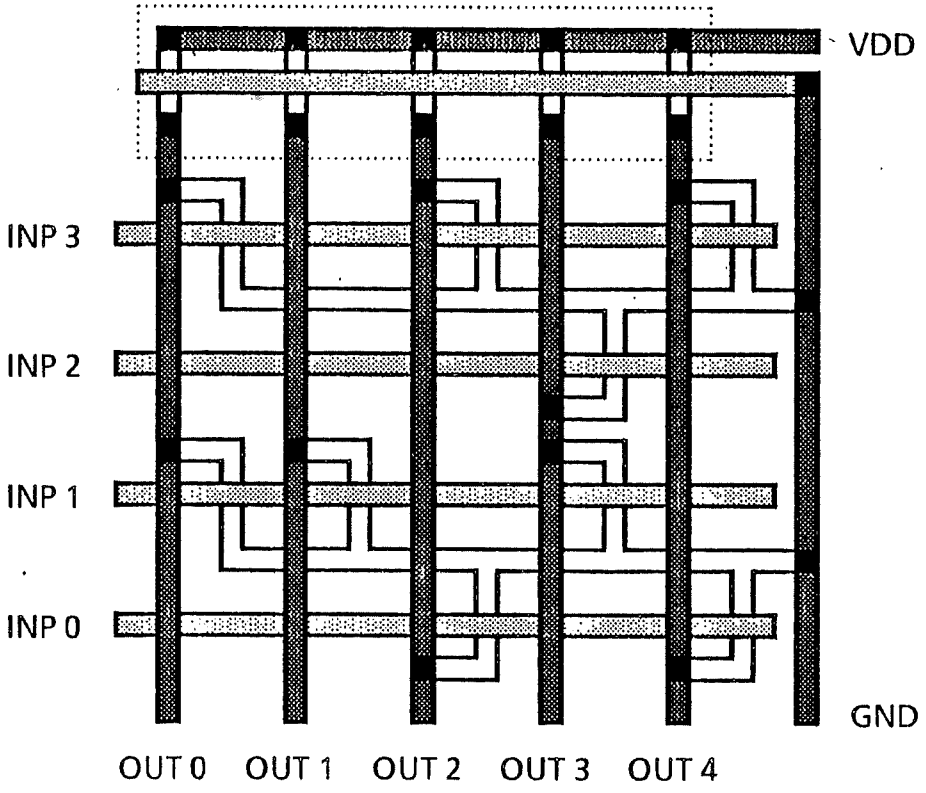

Figure 1: Symbolic Layout for a CMOS NOR Array

of pulldowns in a NOR array. In Section 7 we describe the implementation of an n-bit decoder from a NOR array and a placement function designed for this purpose. Finally, a behavioural specification of a ROM parameterized by a functional specification of its contents is derived from the behaviours of the n-bit decoder and the NOR array.

2 MOS Implementation of NOR Arrays

Figure 1 shows an example of a MOS structure known as a NOR array [Weste85]. Horizontal polysilicon control lines topologically intersect vertical metal bit lines to form an array of grid positions. "Pulldowns" may be placed at various grid positions in the array; the particular placement of pulldowns determines the function implemented by the NOR array. Pulldowns are implemented by N-type transistors connecting the bit line to Gnd. The gate of every pulldown is controlled by the control line running horizontally through the row. Each of the bit lines has a pullup node which will drive the bit line high when none of the pulldowns in the column are selected. When one or more pulldowns in a column are selected, the bit line will be discharged towards Gnd. When high and low are interpreted as logical T and F respectively, each bit line is the logical NOR of those control lines with a corresponding pulldown in the column.

Many of the details which must be considered in the implementation of NOR arrays as fabricated integrated circuits, *eg.* transistor ratios, power dissipation and ground strapping, cannot be modelled here without losing the focus of our presentation. A thorough discussion of NOR array implementations may be found in both [Mead80] and [Weste85].

It is impractical to build a NOR array interactively using a graphics-based design system. However, many CAD systems provide a programming interface which allows "generators" to be written in a high-level programming language, *eg.* ELECTRIC [Rubin83]. For example, a microcode assembler would assemble a microcode specification into a bitmap which would then be used by a ROM generator to place pulldowns in a NOR array implementation of the ROM. We shall see later in this paper how formal verification can be used to correctly implement generators of this kind.

3 Hardware Specification in Higher-Order Logic

[Gordon86] argues that many aspects of digital systems can be formally represented in higher-order logic and that specialized hardware description languages are not needed. Furthermore, the inference rules of higher-order logic provide a practical means of proving systems correct.

The representation of higher-order logic in a computer system is described in [Gordon87]. Several large and interesting examples of hardware verification have been developed using the HOL system including aspects of a proof of correctness for the VIPER microprocessor which involved over one million primitive inferences [Cohn87].

In this paper we use the machine readable notation of the HOL language except for quantifiers where we use the standard logical symbols "$\forall$" and "$\exists$". We also use "ε" and "λ" for Hilbert's choice operator and function abstraction respectively instead of the machine readable symbols. Appendix 1 summarizes the notation of higher-order logic as it is represented in the HOL system.

[Camilleri86] describes how the structure of a hardware design can be represented by a predicate in higher-order logic. Conjunction and existential quantification are used to model composition and internal signal hiding in a hierarchical specification. For example, Definition 1 specifies the implementation of a CMOS inverter from MOS primitives outlined in the next section. Iteration can also be used to describe the structure of hardware devices such as n-bit adders.

Definition 1:

```
⊢ INV (inp:^wire,out:^wire) =
    ∃ w1 w2 w3 w4.
      Vdd (w1) /\ Ptran (inp,w1,w2) /\
        Join (w2,w3,out) /\ Ntran (inp,w4,w3) /\ Gnd (w4)
```

The behaviour of a hardware design can also expressed in higher-order logic by describing a relation between inputs and outputs. In many cases, this will only be a partial specification since it may be unreasonable to describe the behaviour of the device in some conditions, *eg.* invalid inputs. For example, the following theorem describes the behaviour of the CMOS inverter in terms of logical negation when the input is well-defined. The predicate Def and the abstraction function BoolAbs are defined in Section 5.

Theorem 1:

```
⊢ ∀ inp out.
    INV (inp,out)
    ==>
    Def inp ==> Def out /\ (BoolAbs out = ~(BoolAbs inp))
```

4 MOS Level Primitives

The behavioural primitives used in this paper are based on a four value logic with values for high impedance `Zz` and error `Er` in addition to low `Lo` and high `Hi` [Dhingra87]. The four value logic is modelled in the natural numbers; the formal theory includes an axiom that each of the four MOS values are distinct but their actual value in the natural numbers is left unspecified. The values are ordered to form a lattice with a "top", `Er` and a "bottom", `Zz` as shown in Figure 2. Definition 2 defines the least upper bound function based on this ordering.

Definition 2:

```
⊢ x U y = ((x = y) => x | (x = Zz) => y | (y = Zz) => x | Er)
```

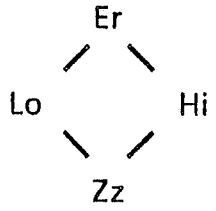

Figure 2: Lattice of MOS Values

The style of hardware specification described in [Camilleri86] models inputs and outputs as time-varying signals. However, in this paper we are concerned only with combinational circuits and so we model inputs and outputs as time-independent values. For example, a wire in the implementation is modelled by a MOS value and a bus is modelled by a function from positions to MOS values

(in effect, a vector of MOS values). When necessary, we can ensure that an n-bit bus is modelled uniquely by forcing all bit positions greater than $n-1$ to some arbitrary but fixed value; the same effect is achieved in [Bowen87] using partial functions with domains restricted to values between 0 and $n-1$. Similarly, a vector of boolean values, *ie.* an n-bit word, is modelled by a function from positions to boolean values. The following abbreviations (using ML variables and anti-quotation) summarize these conventions.

ML Variable	HOL type	Description
`val`	`":num"`	MOS value
`posn`	`":num"`	position
`wire`	`":^val"`	MOS wire
`bus`	`":^posn->^wire"`	n-bit MOS bus
`word`	`":^posn->bool"`	n-bit boolean word

A small collection of MOS primitives represents the only assumptions made about physical devices in this paper. Vdd, Gnd and Float model constant sources of a particular MOS value. The Node primitive models a connecting wire which is useful when an input is connected directly to an output at some stage in a hierarchical specification. The transistor models are uni-directional; even though real transistors are bi-directional they are usually used with a specific orientation that can be determined algorithmically [Clocksin86]. The Join primitive determines the result of merging two wires by taking the least upper bound. Finally, a PullUp primitive models a weakly charging source of Vdd.

Definition 3:
```
⊢ Vdd (o:^wire) = (o = Hi)
```

Definition 4:
```
⊢ Gnd (o:^wire) = (o = Lo)
```

Definition 5:
```
⊢ Float (o:^wire) = (o = Zz)
```

Definition 6:
```
⊢ Node (i:^wire,o:^wire) = (o = i)
```

Definition 7:
```
⊢ Ptran (g:^wire,i:^wire,o:^wire) =
    (o = (g = Lo) => i | ((g = Hi) => Zz | Er))
```

Definition 8:

```
⊢ Ntran (g:^wire,i:^wire,o:^wire) =
    (o = (g = Hi) => i | ((g = Lo) => Zz | Er))
```

Definition 9:

```
⊢ Join (i1:^wire,i2:^wire,o:^wire) = (o = (i1 U i2))
```

Definition 10:

```
⊢ PullUp (i:^wire,o:^wire) = (o = ((i = Zz) => Hi | i))
```

5 Relating MOS Behaviour to Boolean Logic

Informally, MOS designers use boolean logic as an abstraction of voltage values. In this paper, we approximate voltage values with a four value logic. This abstraction consists of mapping the MOS values Lo and Hi to the boolean values false F and true T. This informal mapping from MOS values to boolean values is only partial since the other two possible MOS values, Zz and Er, are ignored. Since all functions defined in higher-order logic must be total, we partially specify a total function where the values Zz and Er are mapped to fixed but unknown boolean values. In particular, BoolAbs may be defined as any function which satisfies Theorem 2. For example, we can define BoolAbs using Hilbert's choice operator, ϵ, as shown in Definition 11 [1]. It is also possible to define BoolAbs using a logical constant for a fixed but unknown boolean value. The corresponding abstraction function for n-bit words simply applies BoolAbs to each bit.

Definition 11:

```
⊢ BoolAbs = εf. (f Lo = F) /\ (f Hi = T)
```

Theorem 2:

```
⊢ (BoolAbs Lo = F) /\ (BoolAbs Hi = T)
```

Definition 12:

```
⊢ WordAbs (b:^bus) = λp. BoolAbs (b p)
```

BoolAbs and WordAbs are used to specify the behaviour of digital devices in boolean logic. For instance, we would like to prove that the inputs and output of a CMOS AND-gate are related by the logical AND function when the inputs and output are viewed abstractly as boolean signals. However, the AND function only describes the behaviour of a CMOS AND-gate when the inputs are well-defined,

[1] "ϵx.P[x]" may be read as the value x satisfying P[x] if such a value exists; otherwise the term denotes an arbitrary value of the correct type. The above definition of BoolAbs illustrates a general method for partially specifying a total function using Hilbert's choice operator.

that is, Lo or Hi. The predicate Def is proposed in [Dhingra87] to describe this condition. We also define Defn which is the n-bit analogue of Def.

Definition 13:

```
⊢ Def (w:^wire) = (w = Lo) \/ (w = Hi)
```

Definition 14:

```
⊢ (Defn 0 (b:^bus) = T) /\
  (Defn (n+1) (b:^bus) = Def (b n) /\ Defn n b)
```

Another level of abstraction is used to map n-bit words to the natural numbers. The abstraction function WordVal interprets an n-bit word as the unsigned binary representation of a number. The definitions of BoolVal and WordVal are based on an example in [Camilleri86].

Definition 15:

```
⊢ BoolVal (b:bool) = (b => 1 | 0)
```

Definition 16:

```
⊢ (WordVal 0 (w:^word) = 0) /\
  (WordVal (n+1) (w:^word) =
    ((2 EXP n) * (BoolVal (w n))) + (WordVal n w))
```

6 Formal Specification of a NOR Array

Both hierarchical decomposition and iteration are used to formally specify the implementation of a NOR array. Hierarchically, a NOR array is implemented by bit columns. In turn, bit columns are implemented by pulldowns and MOS primitives.

As illustrated in Figure 3, the formal specification of a pulldown consists of a N-type transistor, a Gnd node and an instance of the Join primitive. The Join primitive is included in the specification of the pulldown so that it has both an input and an output signal.

Definition 17:

```
⊢ PullDown (cntl:^wire,i:^wire,o:^wire) =
    ∃ w1 w2. Join (i,w1,o) /\ Ntran (cntl,w2,w1) /\ Gnd (w2)
```

A single column in a NOR array is constructed by placing either a connecting wire or a pulldown at each position in the column. The formal specification of a column in a generic NOR array is parameterized by a placement function, "f", which indicates whether a wire or a pulldown is placed at each position in

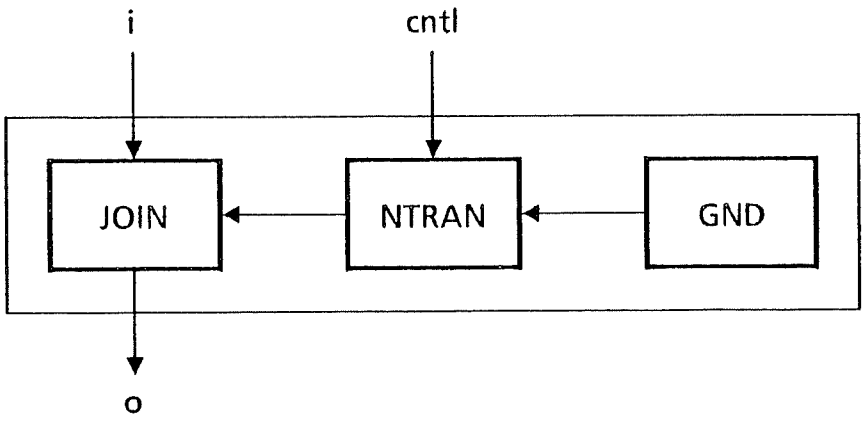

Figure 3: MOS Primitives used to construct a Pulldown Node

the column. A connecting wire is modelled by the Node primitive. It is convenient to use primitive recursion to iterate positions in the bit column due to the interconnection of one position to the next position.

Definition 18:

```
⊢ (NORColumn 0 f (inp:^bus,node:^wire,out:^wire) =
     PullUp (node,out)) /\
   (NORColumn (n+1) f (inp:^bus,node:^wire,out:^wire) =
     ∃w.
       ((f n) => PullDown (inp n,node,w) | Node (node,w)) /\
       NORColumn n f (inp,w,out))
```

As shown in Definition 18, a 0-bit column is simply a pullup node. An $n+1$-bit column is constructed from an n-bit column by the addition of either a connecting wire or a pulldown according the value of the placement function at this position.

A NOR array is constructed by iterating bit columns. In this case, it is easiest to use universal quantification to iterate bit columns because, from a circuit point of view, there are no interconnections from one column to the next column. As shown in Definition 19, it is necessary to "cap" each bit column with a floating node.

Definition 19:

```
⊢ NORArray n m f (inp:^bus,out:^bus) =
     ∀i. (i < m) ==>
       ∃w. Float (w) /\ NORColumn n (f i) (inp,w,(out i))
```

As before, we have parameterized the formal specification of a generic NOR array with a placement function, "f", which supplies the placement function for each column in the array. In this case, "f" is the curried form of a bitmap function

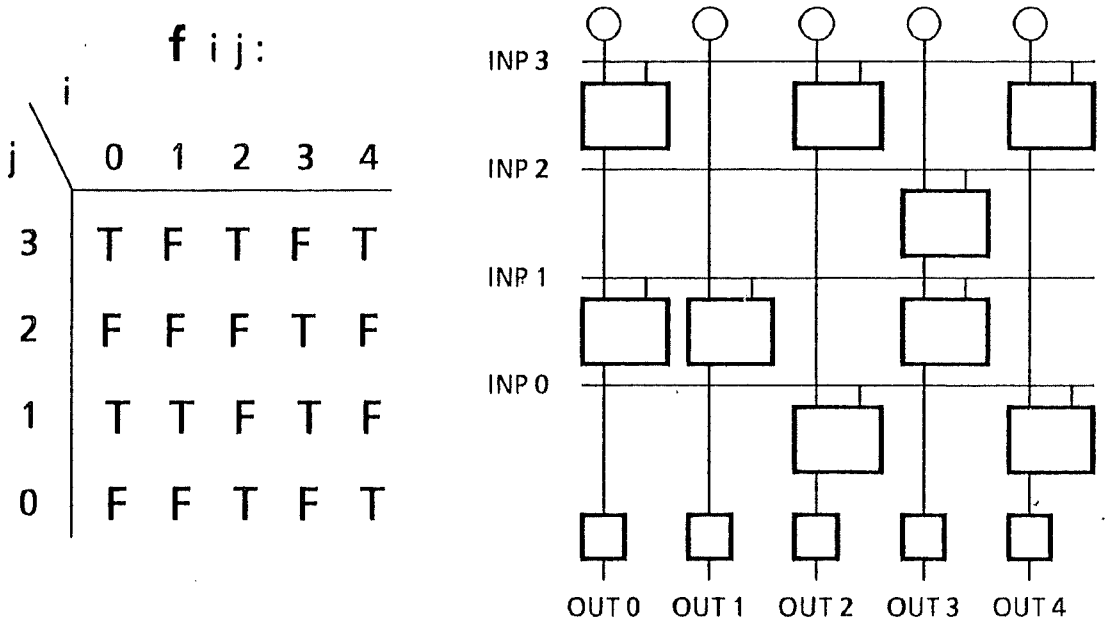

Figure 4: Placement Function and Corresponding NOR Array

which maps array indices to boolean values.

Figure 4 illustrates the construction of a 4 by 5 NOR array from a placement function "f". In contrast to the symbolic layout suggested in Figure 1, the pullup nodes (represented by small boxes in Figure 4) have been moved to the bottom of each column in the formal specification of the NOR array. Our simple model of a pullup makes it necessary to position the pullup device at the output end of the bit column. If none of the pulldowns in the column has discharged the bit line, then the pullup will output the value `Hi`; otherwise, the pullup will simulate the dominant effect of a discharging pulldown by producing the value `Lo` as output. With a more complicated model of MOS behaviour with multiple strengths for `Lo` and `Hi`, it would be possible to more closely follow the symbolic layout shown in Figure 1.

In section 2 we summarized the behaviour of the NOR array by observing that 'when high and low are interpreted as logical `T` and `F` respectively, each bit line is the logical NOR of those control lines with a corresponding pulldown in the column'. We can formalize this description by proving the following theorem. This theorem also takes into account the condition under which this behaviour holds, namely, that the inputs must be well-defined. Two logical operations, the logical OR-ing of all the bits in a single word and the bitwise AND-ing of two n-bit words, are also defined below.

Definition 20:

```
⊢ (ORBits 0 (w:^word) = F) /\
  (ORBits (n+1) (w:^word) = (w n) \/ (ORBits n w))
```

Definition 21:

```
⊢ ANDWords (w1:^word,w2:^word) = λp. (w1 p) /\ (w2 p)
```

Theorem 3:

```
⊢ ∀ n m f.
    ∀ inp out.
      NORArray n m f (inp,out)
      ==>
      Defn n inp ==>
        Defn m out /\
          ∀i. (i < m) ==>
            (WordAbs out i =
              ~(ORBits n (ANDWords (WordAbs inp,f i))))
```

Theorem 3 illustrates one of the major claims in this paper, in particular, that the separation of function from circuit design in regular structures can be exploited to formally derive a generic behaviour parameterized by a functional specification. We have seen how the placement function is used to guide the construction of a particular instance of a NOR array. The placement function also serves as a functional specification for the NOR array. Theorem 3 shows that the behaviour of a particular instance of a NOR array can be determined entirely from its inputs and its functional specification, *ie.* the placement function, without any reference to a circuit implementating the NOR array.

The generation of a NOR array from its functional specification is trivial because the placement of pulldowns in the regular structure corresponds exactly to the functional specification, *ie.* the placement function is just the functional specification. In general, the generation of a regular structure from its functional specification is more complicated. For instance, one would need to write an algorithm to generate a placement function for the AND and OR planes of a PLA from its sum of products equations. Similarly, the implementation of an n-bit decoder by a NOR array depends on the correct design of an algorithm for the placement of pulldowns in the NOR array. From these observations, we tender our second major claim in this paper that the separation of function from circuit design in regular structures shifts the focus of the verification task from circuit details to the correctness of the algorithms used to map a functional specification to an instance of the regular structure.

7 Formal Specification of a Decoder

An n-bit decoder can be implemented by a $2n$ by 2^n NOR array and n inverters. The n wires of the n-bit word and their complements produced by the n inverters are inputs to the NOR array. The placement of pulldowns in the NOR array is

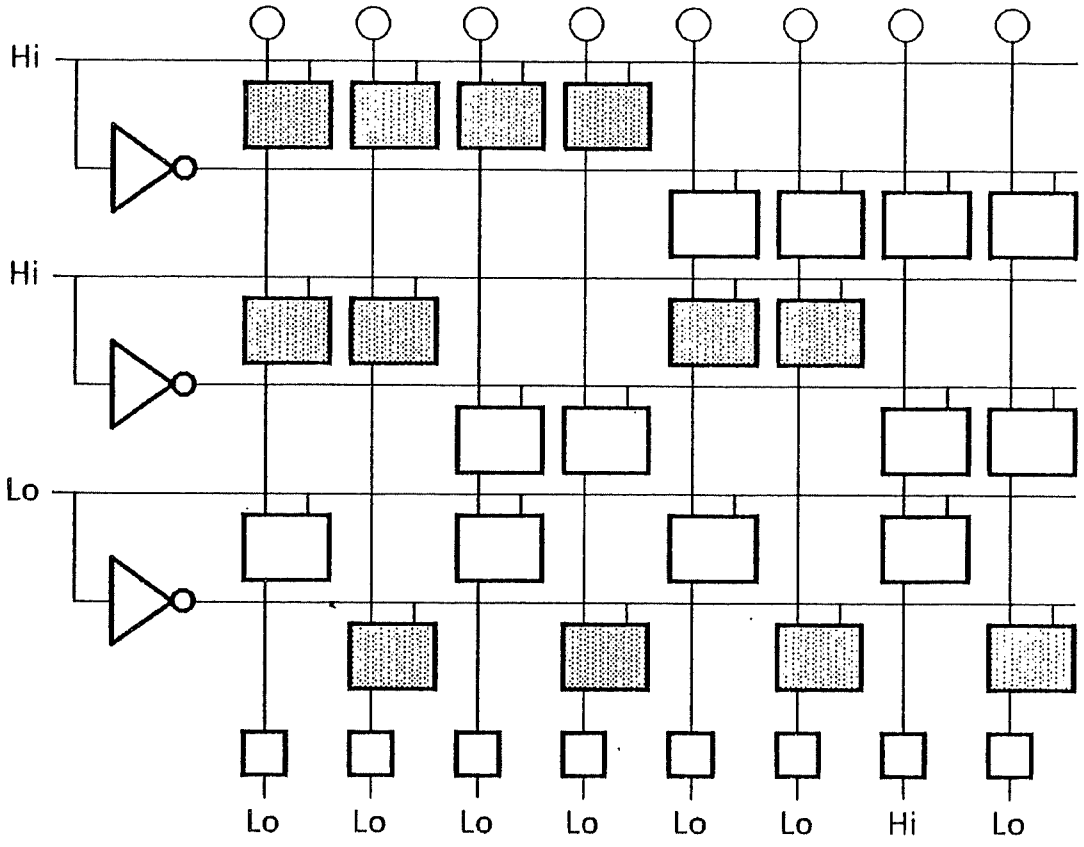

Figure 5: A 3-bit Decoder with the input value "6"

designed to ensure that the ith bit of the output word is high if and only if the input word is the n-bit representation of the number i.

Figure 5 illustrates how the placement of pulldowns by DecoderFunction in a 3-bit decoder has discharged all but the seventh bit line when the input is the value "6". Discharging pulldowns have been shaded in Figure 5.

The placement function for an n-bit decoder, DecoderFunction, is defined in terms of BinRep which is a function that produces the binary representation of number. The definitions of EVEN, EXP, DIV and MOD are given in Appendix 2.

Definition 22:

```
⊢ BinRep n p =
    ~(((n - (n MOD (2 EXP p))) MOD (2 EXP (p+1))) = 0)
```

Definition 23:

```
⊢ DecoderFun i j =
    (EVEN j) => (BinRep i (j DIV 2)) | ~(BinRep i ((j-1) DIV 2))
```

The decoder interface produces $2n$ outputs which alternate between bits of the input word and their complements. The ordering of output bits is tightly coupled with the definition of DecoderFunction. The specification of the decoder interface

and the decoder constructed from the decoder interface and a NOR array are shown below.

Definition 24:

```
⊢ DecoderInterface n (inp:^bus,out:^bus) =
    ∀i. i < (2*n) ==>
       ((EVEN i) =>
         INV (inp (i DIV 2),out i) |
         Node (inp ((i-1) DIV 2),out i))
```

Definition 25:

```
⊢ Decoder n (inp:^bus,out:^bus) =
    ∃b:^bus.
      DecoderInterface n (inp,b) /\
      NORArray (2*n) (2 EXP n) DecoderFun (b,out)
```

The derivation of a behavioural specification for the n-bit decoder depends on a relationship between the functions WordVal and BinRep. WordVal and BinRep are not quite inverses of each other since WordVal only looks at the first n bits of a binary representation. Nevertheless, the two functions are related by the following theorems.

Theorem 4:

```
⊢ ∀ n p. p < n ==> ∀w. BinRep (WordVal n w) p = w p
```

Theorem 5:

```
⊢ ∀ m n. m < 2 EXP n ==> (WordVal n (BinRep m) = m)
```

From these two theorems and the behaviour of a NOR array we can prove the following theorem about the behaviour of the n-bit decoder.

Theorem 6:

```
⊢ ∀n.
    ∀ inp out.
      Decoder n (inp,out)
      ==>
      Defn n inp ==>
        Defn (2 EXP n) out /\
          ∀i. i < (2 EXP n) ==>
            (WordAbs out i = (WordVal n (WordAbs inp) = i))
```

We can paraphrase the above theorem as follows. When the MOS values of the

input word are interpreted as booleans values and then as the n-bit representation of a number, the boolean interpretation of the ith output bit will be T if and only if the input is the number i. This behaviour is constrained by the condition that the inputs must be well-defined. In short, under this condition, the input value i selects the ith output bit.

The proof of this theorem illustrates how the verification task focuses on the algorithm used to generate the regular structure. Arguing for an arbitrary output bit, there are two cases to consider. In one case, the input word has selected this output bit and we must prove that none of the pulldowns in this column of the NOR array has discharged the bit line. In the other case, the input has not selected this output bit and so we must show that at least one of the pulldowns in the bit column has discharged the bit line. Whether or not a pulldown has discharged the bit line depends on its position in the bit column with respect to the current input. Hence, the proof is chiefly concerned with the specific placement of pulldowns in a bit column by DecoderFunction.

8 Formal Specification of a ROM

The implementation of a 4-word, 5-bit ROM by an n-bit decoder and a NOR array is shown in Figure 6. Since the NOR array produces the complement of the addressed word, a row of inverters is included as interfacing logic. These inverters would also restore logic levels and possibly amplify the strength of output signal. Formal specifications for the interfacing logic and the n by m ROM are shown below.

Definition 26:

```
⊢ ROMInterface n (inp:^bus,out:^bus) =
    ∀i. i < n ==> INV (inp i,out i)
```

Definition 27:

```
⊢ ROM n m f (inp:^bus,out:^bus) =
    ∃ b1 b2.
      Decoder n (inp,b1) /\
      NORArray (2 EXP n) m f (b1,b2) /\
      ROMInterface m (b2,out)
```

The behaviour of the ROM follows directly from the derived behaviours of the n-bit decoder and the NOR array. The following theorem states that when the input address is well-defined, then the output of the ROM will be the word at location j in the ROM where j is the numerical value of the input and the contents of the ROM are given by the function "f".

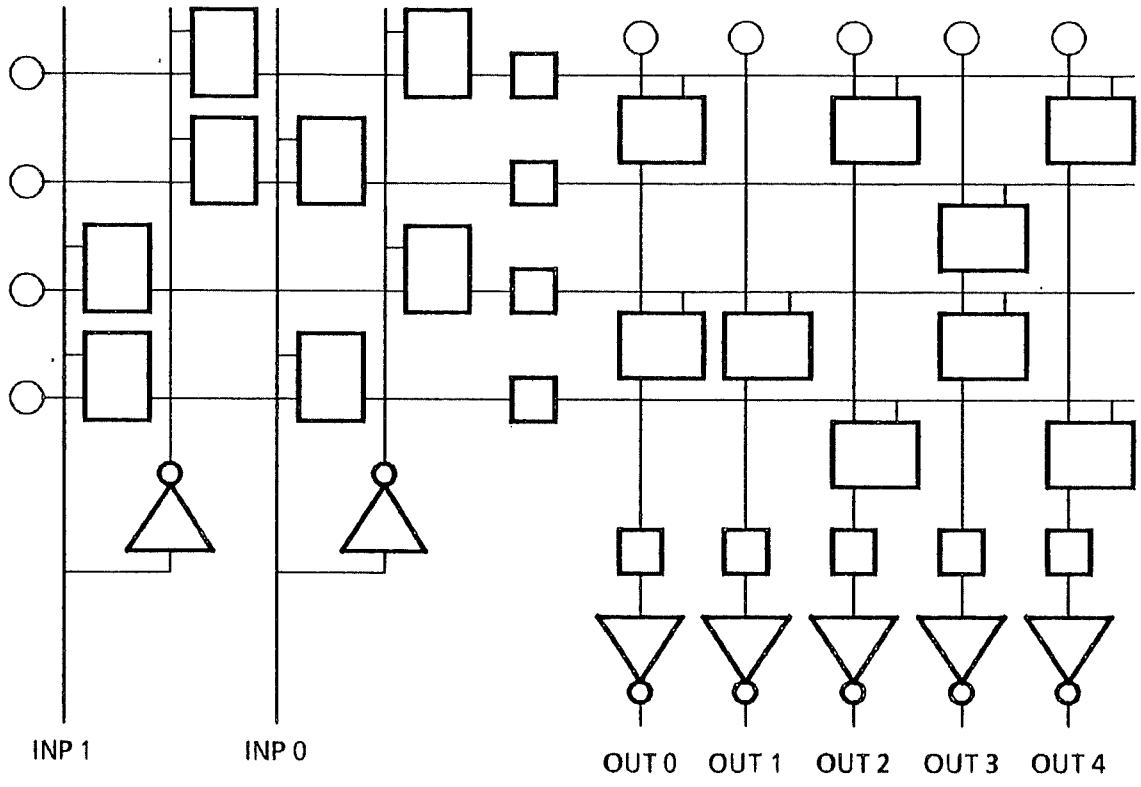

Figure 6: A 4-word, 5-bit ROM implemented from NOR Arrays

Theorem 7:

```
⊢ ∀ n m f.
    ∀ inp out.
      ROM n m f (inp,out)
      ==>
      Defn n inp ==>
        Defn m out /\
          ∀i. i < m ==>
            (WordAbs out i = (f i (WordVal n (WordAbs inp))))
```

Summary and Conclusions

We have described how the use of regular structures such as PLAs and ROMs provides another method for controlling the structural complexity in a hardware specification. More complicated regular structures such as generic data paths might also be used to control structural complexity.

We have also shown how the separation of function from circuit design in a regular structure leads to a behavioural specification based entirely on its functional specification. For example, Theorem 7 relates the behaviour of the ROM

implementation to a functional specification of the ROM contents. In short, for any well-defined input, the output of the ROM can be inferred directly from its functional specification.

The derivation of a behavioural specification for the n-bit decoder shows how the focus of the verification task is shifted to the task of proving the correctness of the algorithm used to generate a regular structure from a functional specification. The most difficult and most interesting aspect of proving the correctness of the ROM implementation is proving the correctness of the algorithm used in the definition of DecoderFunction to place pulldowns in the NOR array of the decoder.

The generation of a PLA or ROM from a functional specification is a simple case of silicon compilation. Hardware verification could be used to validate more complicated silicon compilation techniques especially when a technique is based on some form of regularity. For example, the MacPitts silicon compiler uses a generic floor plan to compile a LISP-like behavioural specification [Siskind82]. Similarly, formal methods presented in this paper could be used to validate aspects of the system described in [Agrawal84] for the synthesis of mask layouts from high-level descriptions of finite state machines.

Finally, we observe that the simplicity of the MOS level primitives, *eg.* unidirectional transistors, does not undermine the more important aspects of the proof. For instance, the correctness of the algorithm used to place pulldowns in the NOR array implementation of the decoder does not absolutely depend on the accuracy of the underlying MOS models. In other words, one of the more important result of this exercise in formal proof is a specification of how to write a program which generates an n-bit decoder through the programming interface of a VLSI CAD system. This result has immediate practical application in a traditional VLSI design context.

References

[Agrawal84] Agrawal, P. and M. Meyer. "Automation in the Design of Finite-State Machines", VLSI Design, Vol. 5, No. 9, September 1984.

[Bowen87] Bowen, J. "The Formal Specification of a Microprocessor Instruction Set", Technical Monograph PRG-60, Computing Laboratory, Oxford University, January 1987.

[Camilleri86] Camilleri, A., M. Gordon and T. Melham, "Hardware Verification using Higher-Order Logic", From HDL Descriptions to Guaranteed Correct Circuit Designs, Proceedings of the IFIP WG 10.2 International Working Conference, Grenoble, France, 9-11 September 1986, D. Borrione, ed., North-Holland, Amsterdam, 1987.

[Clocksin86] Clocksin, W.F. and M.E. Leeser. "Automatic Determination of Signal Flow through MOS Transistor Networks", Integration, Vol. 4, 1986.

[Cohn87] Cohn, A. "A Proof of Correctness of the VIPER Microprocesor: The First Level", VLSI Specification, Verification and Synthesis, Proceedings of the Workshop on Hardware Verification, Calgary, Canada, 12-16 January 1987, G. Birtwistle and P. Subrahmanyam, eds., 1987.

[Dhingra87] Dhingra, I.S., "Formal Validation of an Integrated Circuit Design Style", Proceedings of the Workshop on Hardware Verification, Calgary, Canada, 12-16 January 1987, G. Birtwistle and P. Subrahmanyam, eds., 1987.

[Gordon85] Gordon, M., "HOL: A Machine Oriented Formulation of Higher-Order Logic", Technical Report No. 68, Computer Laboratory, University of Cambridge, 1985.

[Gordon86] Gordon, M., "Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware", Formal Aspects of VLSI Design, Proceedings of the 1985 Edinburgh Conference on VLSI, G.J. Milne and P. Subrahmanyam, eds., North-Holland, Amsterdam, 1986.

[Gordon87] Gordon, M., "A Proof Generating System for Higher-Order Logic", Proceedings of the Workshop on Hardware Verification, Calgary, Canada, 12-16 January 1987, G. Birtwistle and P. Subrahmanyam, eds., 1987.

[Mead80] Mead, C. and L. Conway, Introduction to VLSI Systems, Addison-Wesley, Reading, Massachusetts, 1980.

[Rubin83] Rubin, S., "An Integrated Aid for Top-down Electrical Design", VLSI '83, Proceedings of the IFIP WG 10.5 International Conference on Very Large Scale Integration, Trondheim, Norway, 16-19 August 1983, F. Anceau and E.J. Aas, ed., North-Holland, Amsterdam, 1983.

[Siskind82] Siskind, J., J. Southard and K. Crouch. "Generating Custom High Performance VLSI Designs From Succinct Algorithmic Descriptions", Proceedings of the 1982 Conference on Advanced Research in VLSI, Massachusetts Institute of Technology, 25 January, 1982.

[Weste85] Weste, N. and K. Eshraghian, Principles of CMOS VLSI Design, Addison-Wesley, Reading, Massachusetts, 1985.

Appendix 1

The machine readable syntax of the HOL language is summarized below. There are four kinds of terms in the language: variables, constants, applications (of a function to a term) and function abstractions (lambda expressions).

Higher-order logic extends first-order logic by allowing variables to range over functions and predicates. Such variables are called "higher-order" and can be quantified. Functions and predicates can be arguments and results of other functions and predicates.

Every term in the HOL language has a "type". The use of types prevents inconsistencies (such as Russell's paradox) which would otherwise result from the expressive power gained by allowing higher-order variables. Every type is either atomic or constructed from existing types.

Certain features of the HOL language have a special syntactic status for improved readability, *eg.* definition of infix functions such as EXP, DIV and MOD in Appendix 2.

HOL Syntax	Description
`!x.t`	universal quantification
`?x.t`	existential quantification
`@x.t`	Hilbert's choice operator
`\x.t`	function abstraction
`~b`	negation
`b1 /\ b2`	conjunction
`b1 \/ b2`	disjunction
`b1 ==> b2`	implication
`b => t1 | t2`	conditional expression
`(t1, .., tn)`	n-tuple
`f t1 .. tn`	function application

Appendix 2

The following is a synopsis of definitions and significant theorems used in proving the correctness of the ROM (aside from those given in the paper). Proofs for the well-ordered principle and the division algorithm theorem were supplied by T.Melham. The remaining theorems were proven for this example.

```
Definitions of EXP, DIV, MOD and EVEN
=====================================

EXP = |- (m EXP 0 = 1) /\ (m EXP (SUC n) = m * (m EXP n))

DIV = |- n DIV m = @q. ?r. (n = (q * m) + r) /\ (r < m)

MOD = |- n MOD m = @r. ?q. (n = (q * m) + r) /\ (r < m)

EVEN = |- EVEN n = ((n MOD 2) = 0)


Arithmetic Theorems
===================

WOP = |- !P. (?n. P n) ==> (?n. P n /\ (!m. m < n ==> ~P m))

DA = |- !m n. (0 < n) ==> ?q r. (m = (q * n) + r) /\ r < n

ADD_SUB = |- !m n. ((m + n) - n) = m

ADD_SUB_ASSOC =
|- !m n. m <= n ==> !p. ((p + n) - m) = (p + (n - m))

UNIQUE_QUOTIENT_THM =
|- !m n p. (m < n) /\ (p < n) ==>
     !q s. (((q * n) + m) = ((s * n) + p)) ==> (q = s)

UNIQUE_REMAINDER_THM =
|- !m n p. (m < n) /\ (p < n) ==>
     !q s. (((q * n) + m) = ((s * n) + p)) ==> (m = p)

EXP_ADD_MULT =
|- !m n p. m EXP (n + p) = ((m EXP n) * (m EXP p))
```

Division Theorems
==========

```
EXISTS_DIV =
|- !m. 0 < m ==> !n. ?r. (n = ((n DIV m) * m) + r) /\ (r < m)

DIV_THM =
|- !m n. n < m ==> !k. (((k * m) + n) DIV m) = k

DIV_MULT_LESS_EQ =
|- !m. 0 < m ==> !n. ((n DIV m) * m) <= n

LESS_MULT_DIV_LESS =
|- !m. 0 < m ==> !n p. (n < (p * m)) ==> ((n DIV m) < p)

LESS_DIV_LESS_EQ =
|- !m. 0 < m ==> !n p. (n < p) ==> ((n DIV m) <= (p DIV m))
```

Modulus Theorems
==========

```
EXISTS_MOD =
|- !m. 0 < m ==> !n. ?q. (n = (q * m) + (n MOD m)) /\ (n MOD m) < m

MOD_THM = |- !m n. n < m ==> !k. ((((k * m) + n) MOD m) = n)

MOD_ONE_THM = |- !m. m MOD 1 = 0

LESS_MOD_THM = |- !m n. n < m ==> ((n MOD m) = n)

MOD_LESS_THM = |- !m. 0 < m ==> !n. (n MOD m) < m

MOD_LESS_EQ = |- !m. 0 < m ==> !n. (n MOD m) <= n

MULT_MOD_0 = |- !m. 0 < m ==> !n. ((n * m) MOD m) = 0

MOD_MOD_THM = |- !m. 0 < m ==> !n. ((n MOD m) MOD m) = (n MOD m)

MOD_CONGRUENCE_THM =
|- !m. 0 < m ==> !n k. ((k * m) + n) MOD m = n MOD m
```

```
MOD_ADD_THM =
|- !m. 0 < m ==>
     !n p. (((n MOD m) + (p MOD m)) MOD m) = ((n + p) MOD m)

MOD_SUB_THM =
|- !m n p. (0 < m /\ p <= n MOD m) ==>
     ((((n MOD m) - p) MOD m) = ((n - p) MOD m))

MOD_2_EXP_SUC =
|- !m n. (m MOD (2 EXP n)) <= (m MOD (2 EXP (SUC n)))
```

WordVal and BinRep Theorems

```
WordVal_BinRep_MOD_THM =
|- !m n. WordVal n (BinRep m) = (m MOD (2 EXP n))

WordVal_LESS_EXP = |- !n w. WordVal n w < 2 EXP n

WordVal_MOD_THM =
|- !n p. p < n ==>
     !w. ((WordVal n w) MOD (2 EXP (SUC p))) = WordVal (SUC p) w

WordVal_LESS_OR_EQ_SUC = |- !n w. WordVal n w <= WordVal (SUC n) w
```