AN AUTOMATICALLY VERIFIED GENERALIZED MULTIFUNCTION ARITHMETIC PIPELINE

Matthias Mutz
Universität Passau
Lehrstuhl Rechnerstrukturen (Prof. W. Grass)
D-94030 Passau, Germany
email: mutz@fmi.uni-passau.de

ABSTRACT

The verification of sequential circuits with complex data-paths and non-trivial timing behavior is a difficult task. A multifunctional pipeline is described as an example of such a circuit automatically verified by a verification procedure. The paper aims at presenting a possible target for hardware verification methods both for hardware designers interested in applying such methods and researchers developing such methods.

1. INTRODUCTION

There is much doubt by designers concerning the practicability of current verification systems in hardware design. There are only a few examples of circuits verified on the basis of formal methods, which in principle also (but possibly not in practice) could be validated by simulation. Of course, there are some examples, e.g. [Joy 88]. Indeed, verification methods and simulation often significantly differ, especially when a partial specification is verified or when a correctness proof is established within a formal framework. Verification means to compare the circuit model against a formal specification, while interpreting simulation results means comparing with an informal description. Verification evaluates, in some manner, all possible simulations and compares them with the expected results. A good survey on formal methods concerning hardware verification can be found in [McETu 90]. Most of practicable examples come from approaches using some kind of hardware description language for describing a verification task, e.g. [BoPP 88, VCDeM 90].

Efforts in the direction of a benchmark collection for hardware verification are highly desirable, e.g. [Sta 93]. But there is also the challenge with simulation methods as the most important measure of practicability from a designers point of view. From a designers point of view, the example of a known multifunction pipeline presented here may be interesting because specification and implementation description are given at different levels of abstraction, some kind of timing verification with symbolic times has to be performed, and operand values of the computations to be performed are symbolic, i.e. a complete verification is performed which could not be done by exhaustive simulation. From a ”verifiers” point of view this example may be interesting, because timing conditions as well as heavy data manipulations have to be considered when verifying the correctness.

The verification methodology distinguishes between applying the verification procedure and developing the procedure. The first aspect is highly influenced by the expected demands of chip designers, the second aspect deals with the application of formal methods. In this paper, the first aspect is addressed by presenting the definition of a verification task given as a specification of the expected behavior of a circuit and its implemented behavior. The second aspect is addressed by discussing the problem of providing RT operators for the description of hardware functions.

2 VERIFICATION METHODOLOGY

The verification of logic and switch level circuits with respect to their functional and timing behavior at register transfer (RT) level needs the power of higher-order logic. We have based our developments on the ”HOL prove assistant” [Gor 85]. Unfortunately, it is very difficult and needs a detailed knowledge of the theory behind to establish proofs and to automate this task at least for specific problem classes. For our verification needs, we adopt the deductive approach. On principle, systems based on logic are more powerful, but as a disadvantage it's very difficult to establish a proof and to automate this task. For reasons of practicability, we have followed the following strategy:

1. Providing a hardware description language as user interface seems to be most promising, when aiming at acceptance of verification tools in chip design. The underlying formalism is hidden for the designer. Implementation descriptions are given as hierarchical netlists, which could be supported by a schematic entry. For specifications, the methods of specifying behavior using timing diagrams and computation sequence descriptions are adapted.

2. The task of proving that an implementation description at the register transfer (RT) level meets a specification at a level with more abstract timing aspects is not directly performed in an interactive manner with...
HOL. Instead, it is automatically performed by a program which applies transformation rules to HDL terms in order to partition the task into subtasks of comparing two RT level descriptions. The set of transformation rules is not complete such that the verifier may fail. RT level terms are compared based on efficient BDD representations [Bry 85]. In case of differences in timing or functionality counterexamples are generated by the system.

3. The task partitioning that is performed by the verification program and the transformation rules that derive HDL terms from HDL terms are correct with respect to a formal hardware model. In the style of denotational semantics, HOL terms can be derived from HDL terms. In this way, the correctness of transformations on HDL terms can be proved using HOL [Mut 91a]. But this lies in the responsibility of the developer of the verification program and has only to be done once. Applying the hardware verification program and especially defining a hardware verification task is possible without knowing about the formal background and without knowing HOL.

The practicability of the verification task description method has been turned out by automatically verifying descriptions of synchronous and asynchronous example circuits using a prototype of the verification procedure [Mut 91b]. The circuit presented here is one of these examples. The verification procedure is now part of a high-level synthesis system [GMT 94] verifying the correctness of library modules used by high level synthesis procedures.

3 SPECIFICATION

3.1 Environment

Fig. 3.1 shows the interface and the corresponding HDL block description of the circuit of a generalized pipeline array, which early has been described in [KaSA 74]. Here, a version as described in [Kog 81] is used.

```
TERMINALS phi:binary*4; X:binary;
P:binary*4; A:binary*8;
B,C:binary*6;
RS:binary*13
STRUCTURE Pipe4Imp [dtD] (phi,P,X,A,B,C,
RS(10..13),RS(1..9))
SEQUENCE Specification
RESTRICT
dtD<dtH; dtD<dtL
END
```

Figure 3.1 Interface and HDL description

The circuit "Pipe4" is parameterized by symbolical values declared in the « [...] » part (terms of the HDL are notated enclosed in « »). By default, symbolical values represent natural numbers (non-zero for timing parameters). The external lines, defining the interface of the circuit, are given in the «(...)» part. The description consists of a declaration part declaring external (TERMINALS ...) and internal signals (NODES ...), the implementation description (STRUCTURE...), the specification (SEQUENCE...), and the timing restriction part (RESTRICT...).

The four input vectors "P", "A", "B", and "C" and the input bit "X" are to be combined. The circuit is able to perform one out of four operations in a pipelined manner. What operation has to be performed by the pipeline is specified encoded in "op". The timing of a computation cycle of the pipeline is controlled by four clock signals, where the high phase of a clock phase makes the latches at the corresponding pipeline stage transparent. Transport delays of the latches are considered. The clock bits are collected in a four digit vector "phi". "R" and "S" are the output signals, here collected in a vector "RS".

The behavior of the implementation can be defined in a hierarchical manner by structural refinement and is here hidden in the block "Pipe4Imp" (more details in section 4).

The specification of the expected behavior strongly depends on the clocking scheme. It is convenient to restrict the specification of the expected behavior in a specific environment, i.e. assuming a specific timing behavior of the input signals. Figure 3.2 illustrates a simplified behavior of the environment not covering the possibility of overlapping computation cycles. Furthermore, a sketch of the HDL specification is given.

```
A B C X P
phi(1)
phi(2)
phi(3)
phi(4)
RS

1) symbolic constants
2) f (symbolic constants)
```

Figure 3.2: specification of the pipeline circuit
The specification of the expected behavior covers informations about the input values, the clocking scheme (duration dtD between two successive high phases and high phase duration dtH), and the expected output function when the fourth high phase occurs. Furthermore, a delay dtD for the latches, which separate the pipeline stages, is taken into consideration. Which operation has to be performed is encoded in the code "op", where «#op#*2» denotes the 2-bit vector encoding the natural number represented by parameter "op". In logical constants the most significant bit is written most left, e.g. c(4..1) = '1000', c(4)='1', c(3..1)=000'. Operation and values are specified symbolically, i.e. the verification task is performed for all possible operations and values. When the fourth high phase occurs, the final latch stage becomes transparent, and - after the latch delay dtD - results depending on the operation code are expected at RS.

3.2 Data formats and arithmetical operations

The first operation with operation code '00' is defined for the data format given in Figure 3.3. It adds A to the product of B and P:

\[ \text{num}(S(9..3)) = (\text{num}(A(8..3)) + [\text{num}(B(6..3)) \times \text{num}(P(3..1))]) \mod 2^7. \]

The natural number binary encoded in the bits m to n of signal x isnotated \text{num}(x(m..n)), e.g. if A(8..1)='10100110', then \text{num}(A(8..3)) represents the natural number binary encoded by '1010': 41. Only seven bits of the results are significant, i.e. an overflow will not be detected.

The second operation with operation code '01' is defined for the data format given in Figure 3.4. It computes the square of P:

\[ \text{num}(S(8..1)) = \text{num}(P(4..1))^2. \]

The third operation with operation code '10' is defined for the data format given in Figure 3.5. It devides A by B, so that \text{num}(R(3..1)) = \text{num}(A(8..3)) \div \text{num}(B(6..3)) and \text{num}(S(6..3)) = \text{num}(A(8..3)) \mod \text{num}(B(6..3)) \text{ hold.} \text{num}(B(6..3)) \geq 2^3 \text must hold.}

As a HDL feature, common subexpressions may be specified only once. An identifier is assigned to a subexpression (e.g. "DIV") which can be used more than once just by referring to this identifier (e.g. <DIV>). "Vdivmod" devides an m-bit vector by an n-bit vector with the (m-n+1)-bit result and the n-bit rest resulting in a common (m-1)-bit vector.

The fourth operation with operation code '11' is defined for the data format given in Figure 3.6. It builds the square root from A:

\[ \text{num}(R(4..1)) = \sqrt{\text{num}(A(8..1))} \text { and } \text{num}(S(6..3)) = \sqrt{\text{num}(A(8..1))}, \]

with

\[ x = \left( \frac{\text{div}}{\sqrt{x}} \right)^2 + \frac{\text{mod}}{\sqrt{x}} \text{ and } \left( \frac{\text{div}}{\sqrt{x}} + 1 \right)^2 > x. \]
The semantic definition of terms allows to transform a term \texttt{\langle Vmult:a,b \rangle} into a HOL term \texttt{"Vmult:signal\rightarrow\text{num}"}, defining the RT level function in terms of the formal behavioral model of circuits. \texttt{"veclen:signal\rightarrow\text{num}"} maps a signal to its non-zero vector length. The decoding function \texttt{\langle lognum:value \rangle} is defined in a HOL theory that determines the natural number encoded in a bit vector, which is a result of a signal function at a specific time instant. The multiplication \texttt{\langle \times:value\rightarrow\text{num} \rightarrow\text{num} \rightarrow\text{num} \rangle} is defined for natural numbers. The natural number represented by a value of a signal \texttt{"in:value"} is then defined by \texttt{\langle lognum \rangle \langle in \rangle}. The proof, that a term \texttt{\langle Vmult:a,b \rangle} represents for all time instants \( t \) the number that is obtained after multiplying the numbers represented by \texttt{\langle a \rangle} and \texttt{\langle b \rangle} at time \( t \) is established by proving

\[
\vdash \text{((m = veclen a) \land (n = veclen b) \land n > 0)} \supset
\forall \ t.\ \text{lognum } ((\text{Vmult } [m;n] \ [a;b])\ t) =\n\text{lognum } (a\ t) \times \text{lognum } (b\ t)
\]

The proof is not easy to construct and a complete presentation of the prove would go beyond the scope of this paper. The main task is to prove the following lemmas:

\[
\vdash \text{(n > 0)} \supset \forall \ t.\ \text{lognum } ((\text{Vmult } [m;n] \ [a;b])\ t) =\n\text{lognum } (Z\ (n-1)\ n) \times \text{lognum } (b\ t)
\]

where the primitive recursive definition of \( Z \) corresponds to the operator definition with arithmetical interpretation of the operations on bitvectors (this has to be proved as an additional lemma):

\[
\text{Z 0 n x y = if } \text{(even y) then 0 else } x \times 2^{(n-1)}
\]

\[
\text{Z (i+1) n x y = if } \text{(even } y \text{ div } 2^{(i+1)})\text{ then }\ (Z \text{in} \ x \ y) \text{ div } 2\text{ else } (x + (Z \text{in} \ x \ y) \text{ div } 2^{n-1}) + ((Z \text{in} \ x \ y) \text{ mod } 2^n) \text{ div } 2
\]

### 4 IMPLEMENTATION DESCRIPTION

The circuit is composed of two basic components (Fig. 4.1). The HDL descriptions are straightforward.
The computation cells of a pipeline stage are combined to form a computation stage.

```plaintext
BLOCK CompStageImp [n, dtD] (phi, Xin, Fin, Cout, A, B, C, S, D, E)
TERMINALS
  phi, Xin, Fin, Cout: binary
  A, B, C, S, D, E: binary*n
END

NODES
  SS, DD, EE: binary*n
  XX, FF: binary*n
  phi, Xin, Fin, Cout: binary

STRUCTURE
  Connect [1] (XX(1), CC(1));
  Connect [1] (Fin, FF(n+1));
  Connect [1] (Xin, XX(n+1));
  Latch [n, dtD] (phi, DD, D);
  Latch [n, dtD] (phi, SS, S);
END

BEHAVIOR
  CompCell (A(i), B(i), C(i), XX(i+1), FF(i+1), CC(i),
            DD(i), EE(i), SS(i), XX(i), FF(i), CC(i+1))
  Connect [1] (Fin, FF(n+1));
  Connect [1] (CC(n+1), Cout);
  Connect [1] (Xin, XX(n+1));
STRUCTURE
  XX, FF, CC: binary*(n+1)
  SS, DD, EE: binary*n
  phi, Xin, Fin, Cout: binary
  A, B, C, S, D, E: binary*n

NODES
TERMINALS
BLOCK CompStageImp [n, dtD] (phi, Xin, Fin, Cout, A, B, C, S, D, E)
TERMINALS
  phi, Xin, Fin, Cout: binary
  A, B, C, S, D, E: binary*n
END

NODES
  SS, DD, EE: binary*n
  XX, FF: binary*n
  phi, Xin, Fin, Cout: binary

STRUCTURE
  Connect [1] (XX(1), CC(1));
  Connect [1] (Fin, FF(n+1));
  Connect [1] (Xin, XX(n+1));
  Latch [n, dtD] (phi, DD, D);
  Latch [n, dtD] (phi, SS, S);
END

BEHAVIOR
  CompCell (A(i), B(i), C(i), XX(i+1), FF(i+1), CC(i),
            DD(i), EE(i), SS(i), XX(i), FF(i), CC(i+1))
  Connect [1] (Fin, FF(n+1));
  Connect [1] (CC(n+1), Cout);
  Connect [1] (Xin, XX(n+1));
STRUCTURE
  XX, FF, CC: binary*(n+1)
  SS, DD, EE: binary*n
  phi, Xin, Fin, Cout: binary
  A, B, C, S, D, E: binary*n

NODES
TERMINALS
```

The previous block defines the connection of \( n \) computation cells to build up the computation part of a pipeline stage. It contains both a structural description and a behavioral description (equivalence automatically checked).

The combination with "LLAST (phi)" defines that the value of "out" is equal to the value of "in" if "phi" is 1, and it doesn't change if "phi" is 0. The combination with "TPRED [dtD]" defines a transport delay of \( dtD \) time and it doesn't change if "phi" is 0. The combination with value of "out" is equal to the value of "in" if "phi" is 1, behavior description (equivalence automatically checked).

In order to cover all possible sequences, a verification graph is maintained. In this example, we have a very simple timing behavior with just one possible sequence. The created verification graph - it is actually protocolled. In this example, we have a very simple timing behavior with just one possible sequence. The created verification graph - it is actually protocolled. In this example, we have a very simple timing behavior with just one possible sequence. The created verification graph - it is actually protocolled. In this example, we have a very simple timing behavior with just one possible sequence.

### 5.2 Performing the verification task

The verification procedure determines all possible sequences of assignments of terms to signal variables by determining sequences of restricted implementation descriptions [Mut 91b]. Possible sequences of time instants, in which restrictions occur, depend on the specification, which could define several possible orders, and on the delay times from the implementation description.

In order to cover all possible sequences, a verification graph is maintained. In this example, we have a very simple timing behavior with just one possible sequence. The created verification graph - it is actually protocolled in a textual form - is given in Figure 5.1.

Figure 5.1: generated verification nodes

The first restriction of the output signal "RS" occurs at time instant \( t_0 + 4\times dtL + 3\times dtH + dtD \). For the same time instant an expected output function is specified. The comparison of the two descriptions shows, that the two terms represent the same function.

### 6 CONCLUSIONS

This example has demonstrated, that automatic verification is possible for circuits with non-trivial datapaths and timing. Providing operator definitions is a difficult task, because the correctness of the definition has to be proved with respect to an arithmetical interpretation. Operator definitions can be optimized with respect to efficiency of BDD construction, because different definitions for the same function lead to the same BDD representations. The
operator definition is independent from a concrete verification task.

It has been shown, how the behavior of the environment is taken into consideration in the specification. Here, a simplified behavior of the environment was assumed, where overlapping computation cycles have not been taken into consideration. That is the reason, why performing the verification procedure recalls symbolic simulation. It is to mention, that the verification procedure is much more than symbolic simulation: it covers the restriction of signals to functions (and not only to values), it covers the possibility of specifying several possible orders of time intervals and the consideration of timing restrictions, and its term manipulations are based on formally verified transformation rules (correct with respect to formal semantics of the HDL and a formal circuit model).

REFERENCES


[Sta 93] Jørgen Staunstrup (ed.): "IFIP WG 10.2 Collection of Circuit Verification Examples," Department of Computer Science, TU Denmark, Dec. 7 1993


APPENDIX

OPERATOR Vadd [n] (a, b, carry) RESULT

OPERATIONS Vadd:binary*3

NODES a,b,carry:binary*1

BEHAVIOR

{s[1],b[n]}:=(Vadd:s[1],b[n])

END;