N94-18371

Formal Specification of a High Speed CMOS Correlator

P. J. Windley
Department of Computer Science
University of Idaho
Moscow, ID 83843
208.885.6501

Abstract: **The formal specification of a high speed CMOS correlator is presented. The specification gives the high–level behavior of the correlator and provides a clear, unambiguous description of the high–level architecture of the device.**

1 Introduction.

The use of formal specification in designing VLSI circuits has many benefits. Perhaps the most important result is a clear description of the design's behavior that can be used for communication among design engineers, production engineers, test engineers, technical writers, and, perhaps most importantly, customers. Formal specifications also provide a firm foundation upon which analysis of the circuit design can take place. This analysis has the potential to significantly reduce design errors as well as providing a basis for demonstrating that the design has desired properties.

This paper presents the formal specification of a high–speed CMOS correlator [2]. The correlator, which is designed to be used in a space–born spectrometer, contains 32 channels and is capable of sampling at 25MHz.

2 Formal Specification and Verification.

VLSI devices can be specified at many levels of abstraction [8]. Generally, we need at least a behavioral and a structural specification [4]. The behavioral specification is written in logic and unambiguously describes the expected behavior of the device. The behavioral specification is declarative rather than imperative, giving a clear relationship between the inputs, current state, and outputs.

The structural specification describes, again using logic, how the circuit is put together. Ideally, the structural specification can be derived from design information captured by conventional CAD tools or translated from a hardware description language such as VHDL [6].

Verification is nothing more than a mathematical analysis of the behavioral and structural models. Ideally, we would like to show that the intended behavior follows from the structure. This analysis, which is a type of symbolic simulation, can be done by hand or with the aid of mechanical verification tools [5]. These mathematical models can also be used to analytically demonstrate selected behavioral properties for a computer system.

3 A Brief Introduction to HOL.

To formally model hardware and to ensure the accuracy of our proofs, we felt that it was necessary to develop the proofs and properties using a mechanical verification system. This prevents proofs from containing logical mistakes, and assures that the foundations on which the work is based are sound. Due to the nature of the proofs, which include quantification over sets of objects, we felt that a system which supports higher-order logic and a typed lambda calculus would facilitate our efforts. The HOL system was selected for this project due to its support for higher-order logic, generic specifications and polymorphic type constructs. Furthermore its availability, ruggedness, local support, and a growing world-wide user base made it a very attractive selection. In this section we will provide a brief description of HOL.

HOL is a general theorem proving system developed at the University of Cambridge [5,1] that is based on Church's theory of simple types, or higher-order logic [3]. Although Church developed higher-order logic as a foundation for mathematics, it can be used for reasoning about computational systems of all kinds. Similar to predicate logic in allowing quantification over variables, higher-order logic also allows quantification over predicates and functions thus permitting more general systems to be described.

HOL is not a fully automated theorem prover but is more than simply a proof checker, falling somewhere between these two extremes. HOL has several features that contribute to its use as a verification environment:

1. Several built–in theories, including booleans, individuals, numbers, products, sums, lists, and trees. These theories build on the five axioms that form the basis of higher-order logic to derive a large number of theorems that follow from them.

2. Rules of inference for higher-order logic. These rules contain not only the eight basic rules of inference from higher–order logic, but also a large body of *derived* inference rules that allow proofs to proceed using larger steps. The HOL system has rules that implement the standard introduction and elimination rules for Predicate Calculus as well as specialized rules for rewriting terms.

3. A large collection of tactics to support goal directed proof. Examples of tactics include `REWRITE_TAC` which rewrites a goal according to some previously proven theorem or definition, `GEN_TAC` which removes unnecessary universally quantified variables from the front of a goal, and `EQ_TAC` which says that to show two things are equivalent, we should show that they imply each other.

4. A proof management system that keeps track of the state of an interactive proof session.

5. A metalanguage, ML, for programming and extending the theorem prover. Using the metalanguage, tactics can be put together to form more powerful tactics, new tactics can be written, and theorems can be aggregated to form new theories for later use. The metalanguage makes the verification system extremely flexible.

Operator	*Application*	*Meaning*
=	t1 = t2	t1 equals t2
,	t1,t2	the pair t1 and t2
$\wedge$	t1 $\wedge$ t2	t1 and t2
$\vee$	t1 $\vee$ t2	t1 or t2
$\Longrightarrow$	t1 $\Longrightarrow$ t2	t1 implies t2

Table 1: HOL Infix Operators

Binder	*Application*	*Meaning*
$\forall$	$\forall$ x. t	for all x, t
$\exists$	$\exists$ x. t	there exists an x such that t
ε	ε x. t	choose an x such that t is true

Table 2: HOL Binders

In the HOL system there are several predefined constants which can belong to two special syntactic classes. Constants of arity 2 can be declared to be infix. Infix operators are written "rand1 op rand2" instead of in the usual prefix form: "op rand1 rand2". Table 1 shows several of HOL's built–in infix operators.

Constants can also belong another special class called binders. A familiar example of a binder is $\forall$. If c is a binder, then the term "c x.t" (where x is a variable) is written as shorthand for the term "c(λ x. t)". Table 2 shows several of HOL's built–in binders.

In addition to the infix constants and binders, HOL has a conditional statement that is written a $\rightarrow$ b | c, meaning "if a, then b, else c."

4 The Correlator Design.

The correlator is designed for a space borne spectrometer. The design accepts two 2–bit data streams clocked at a maximum of 25MHz. Delayed versions of one stream are multiplied (using a biased multiplication) with the undelayed signal on the other stream. The products are accumulated. The process continues for the duration of the integration cycle which is defined by the int control line. When the end of an integration period is signaled, the results are latched into a register, the accumulators are cleared, and the datardy line goes high to signal that data is ready to be read from the chip. A new integration cycle can begin immediately. Concurrent with the new integration period, the data from the previous integration period can be read on the output lines. Data is read in either a word serial or byte serial mode depending on the value of a control line.

Readers interested in additional detail are referred to [2].

5 The Correlator Specification.

This section presents the behavioral specification of the correlator.

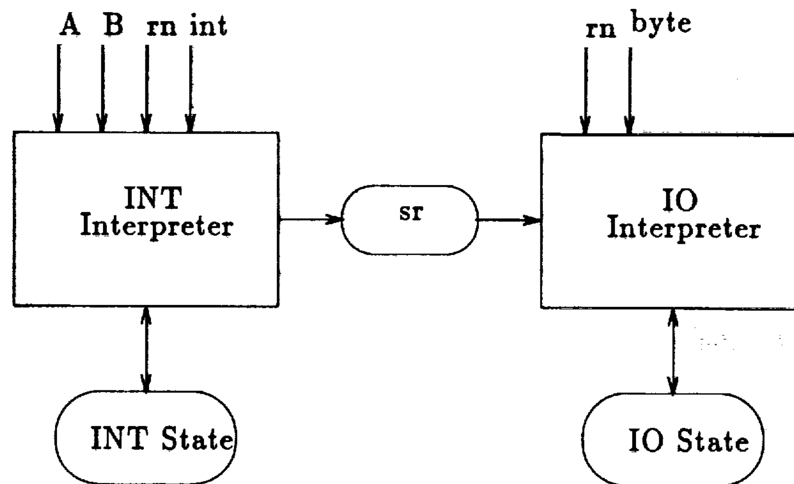

Figure 1: Architecture of the correlator shows the producer—consumer relationship between the **INT** interpreter and the **IO** interpreter.

The overall architecture of the behavioral description is shown in Figure 1. The architecture is based on two separate state machines which, along with the datapath, function as single instruction interpreters [7]. The interpreters are arranged in a producer–consumer architecture with a register serving as the shared link between the two interpreters.

The producer portion of the design is the **INT** interpreter. **INT** performs the integration of the incoming signals in 32 channels. The interpreter controls the following state variables:

- **acc**—A bank of 32, 4–bit accumulators.
- **delay**—A bank of 32, 2–bit delay elements.
- **sr**—A bank of 32, 24–bit shift registers.
- **count**—A bank of 32, 24–bit counters.

Each of these state variables is parameterized for time and channel number and has type **:time→num→**w, where w varies with register width.

The specification for **INT** relates the state variables at time $t+1$ to the their value at time t and the value of the inputs at time t.

```
⊢def integrate_int (acc, delay, sr, count, datardy)
                (a, b, int, rn) =
    ∀t.
    let nextstate = ((int t) →  integrate | dump) = (
    (acc (t+1), delay (t+1), sr (t+1), count (t+1), datardy (t+1)) =
       nextstate (acc t, delay t, sr t, count t, datardy t)
                 (a t, b t, int t, rn t))
```

The function `nextstate` evaluates to either `integrate` or dump depending on the value of the `int` line.

The individual instructions produce new values for the state variables. In the case of the `integrate` instruction new values are calculated for the **acc**, **delay**, and **count** variables. The shift register (`sr`) is unchanged.

```
⊢def integrate (acc, delay, sr, count, datardy)
             (a, b, int, rn) =
    let signal_product n = mapper (delay n) b in (
    let new_acc n  =
        rn →  (bt4_ival 0) |
              (add4 (signal_product n, acc n)) in
    let new_delay n = (n=0) →  a | (delay (n-1)) in
    let new_count n =
       rn                                   →  (wordn 0) |
       (carry4 (signal_product n, acc n)) →  inc (count n) |
                                              (count n) in
    (new_acc, new_delay, sr, new_count, datardy)
```

The new values are precisely described. For example, the new value of the nth accumulator is calculated by adding a biased multiplication of the n–delayed signal and the undelayed signal to the current value in the same accumulator.

The consumer portion of the circuit is the `IO` interpreter. The interpreter controls the following state variables:

- `sr`—A bank of 32, 24–bit shift registers. This is the same register as the `sr` register in the `INT` interpreter.
- `counter`—A 7–bit counter for counting the output.
- `out`—A 16–bit register that latches the values on the output lines.
- `borw`—A state variable that indicates whether output is byte or work serial.

The specification for the `IO` interpreter is similar to the specification of the `INT` interpreter. The `IO` interpreter has six instructions. The interpreter can be reset, it can start the read cycle, it can end the read cycle, it can dump data from the output registers a byte at a time, it can dump data a word at a time, or it can do nothing.

```
⊢def io_int (sr, counter, out, borw, datardy, begin)
          (byte, rn, outck) =
     ∀ t .
     let nextstate =
       ((rn t)                                    →  reset        |
        ((datardy t) ∧ begin                       →  start_read   |
        ((datardy t) ∧ ((val (counter t)) = 0)) →  end_read     |
        ((datardy t) ∧ (borw t) ∧ (outck t)      →  dump_byte    |
        ((datardy t) ∧ ¬(borw t) ∧ (outck t)     →  dump_word    |
                                                     noop        ) in (
     (sr (t+1), counter (t+1), out (t+1), borw (t+1), datardy (t+1)) =
        nextstate (sr t, counter t, out t, borw t, datardy t, begin t)
                  (byte t, rn t, outck t))
```

The operation of IO is more complicated that the operation of INT. Whenever the reset line is raised, the state is reset as described in the specification of the reset operation. When the datardy line goes high, the interpreter begins a read cycle. When the outck line is raised and the datardy line is high, we dump either bytes or words depending on the value of the borw line. There is a counter so that the correct number of bytes and words are dumped. When the counter reaches 0 we end the read cycle (by pulling the datardy line low). Otherwise, we do nothing. [1]

As an example of the instructions in IO, consider the dump_word instruction.

```
⊢def dump_word (sr, counter, out, borw, datardy, begin)
           (byte, rn, outck) =
     let new_counter = (dec counter) in
     let i = (val counter) in
     let new_out = short (sr i) in
     (sr, new_counter, new_out, borw, datardy, begin)
```

The instruction updates the counter by decrementing the old value. The value on the output is determined by 16 most significant bits from the ith shift register, where i is the value of the counter.

The most interesting feature of the specification of INT and IO is that they share state. For example, both specify changes to sr, the variable representing the shift register. INT produces a value that is placed in sr by its dump instruction. IO uses that value when asked to present the results of the integration on the output lines.

Both interpreters also specify changes to datardy, the variable representing whether or not data is ready to be output. INT sets datardy when it has dumped the contents of the the counter into the shift register. IO resets datardy when it is done outputting the data.

Readers of this specification who are familiar with the design may be surprised to find that some details in the circuit are not found in the specification. For instance, after the end of the integration period ends, there is an 8 cycle delay before data can be read from

[1] Note that count in INT and counter in IO are two different state variables.

the chip (i.e. datardy goes high). In the specification shown above datardy goes high the time period after the int line is pulled high. This is an example of the temporal abstraction going on between the circuit levels of the specification and the behavioral specifications given here.

6 The Top–Level Specification

The final specification combines the specifications of the two interpreters and operates them in parallel.

```
⊢def corr_top rep (acc, delay, sr, count, datardy,
                   begin, counter, out, borw)
                  (byte_e, rn, outck, a, b, int) =
    ((integrate_int rep (acc, delay, sr, count, datardy)
                        (a, b, int, rn)) ∧
     (io_int rep (sr, counter, out, borw, datardy, begin)
                  (byte_e, rn, outck)))
```

The specification does not explicitly answer questions regarding the shared use of the sr and datardy lines. For example, do INT and IO correctly coordinate the writing and reading of sr correctly? This and other important questions regarding the operation of the correlator can be answered by analysis of the specification.

7 Conclusion.

This paper has presented the behavioral specification for a VLSI correlator design. Previous to this specification being written, the design was described in design documents and papers such as [2]. These descriptions were necessarily ambiguous since they were written in English. Deriving the specification by reading the design documents and talking to the design engineer provides an interesting perspective on the design process. The behavioral specification of the correlator documents the design and is useful for enhancing communication between designers, customers, and users by unambiguously describing the function of the device.

The specification presented in this paper is a snapshot of the design. A specification is constantly subject to revision to bring it up to date with current expectation and to correct errors that are part of any written description. Future work will extend the specification in two ways:

- We intend to show that the specification meets certain requirements for correct operation. For example, the analysis will make explicit the synchronization conditions that must exist between the two interpreters for the chip to function correctly and show that they are met.

- We will specify the structural level by deriving it from the design information captured in the HDL description of the circuit. We intend to show that this structural specification implies the architecture we have described above.

Acknowledgments

This work was sponsored by NASA under Space Engineering Research Center grant NAGW-1406.

References

[1] Albert Camilleri, Mike Gordon, and Tom Melham. Hardware verification using higher order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*. Elsevier Scientific Publishers, 1987.

[2] J. Canaris and S. Whitaker. A high speed CMOS correlator. In *NASA Space Engineering Research Center Symposium on VLSI Design*, pages 3.3.1–3.3.11, November 1990.

[3] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5, 1940.

[4] Michael J.C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177. Elsevier Scientific Publishers, 1986.

[5] Michael J.C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwhistle and P.A Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*. Kluwer Academic Press, 1988.

[6] IEEE Std 1076-1987. *IEEE Standard VHDL Language Reference Manual*, 1987.

[7] Phillip J. Windley. *The Formal Verification of Generic Interpreters*. PhD thesis, University of California, Davis, Division of Computer Science, June 1990.

[8] Phillip J. Windley. A hierarchical methodology for the verification of microprogrammed microprocessors. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1990.