Synthesis of Synchronous Assertions with Guarded Atomic Actions

Michael Pellauer[†], Mieszko Lis[‡], Donald Baltus[‡], Rishiyur Nikhil[‡]

[†]Massachusetts Institute of Technology [‡]Bluespec, Inc.
pellauer@csail.mit.edu, {elf, baltus, nikhil}@bluespec.com

24th May 2005

Abstract

The SystemVerilog standard introduces SystemVerilog Assertions (SVA), a synchronous assertion package based on the temporal-logic semantics of PSL. Traditionally assertions are checked in software simulation. We introduce a method for synthesizing SVA directly into hardware modules in Bluespec SystemVerilog. This opens up new possibilities for FPGA-accelerated testbenches, hardware/software co-emulation, dynamic verification and fault-tolerance. We describe adding synthesizable assertions to a cache controller, and investigate their hardware cost.

1 Introduction

As modern cores grow increasingly complex, design teams are finding that ad-hoc verification approaches do not scale. Modern hardware synthesis has reached the point where 25% of time is typically spent on design, and 75% on verification. To alleviate this designers are increasingly turning to Assertion-Based Verification [8]. The PSL standard [1] has demonstrated the power of temporal logic for synchronous hardware verification. The SystemVerilog standard [2] introduces SystemVerilog Assertions (SVA), which use the same underlying semantics as PSL.

Traditionally assertions are verified in software simulation. Previous projects such as FoCs [7] have introduced the ability to translate assertions into synthesizable RTL code. This capability offers new verification possibilities. For example, a designer may want to synthesize the entire testbench and device-under-test to an FPGA. Assertions would be checked in hardware, leaving only the test data to be loaded by software. Normally assertions are removed before committing a design to silicon, but designers may want to leave synthesized assertions in the final device for additional dynamic verification or fault tolerance.

We present a method for synthesizing SVA temporal logic assertions into hardware modules in Bluespec SystemVerilog (BSV). Traditional assertion synthesis applies temporal logic techniques directly to explicit clock signals. BSV, in contrast, is an unclocked model that is translated into sequential hardware by means of a scheduler. The main contribution of this paper is a method for translating synchronous assertions into unclocked guarded atomic actions, without interfering with the schedule of the device-under-test.

In this paper we present an overview of SystemVerilog Assertions. We review the Bluespec semantic model, which is unclocked, and investigate the role assertions could play in a BSV project. We demonstrate that a significant subset of the language is efficiently synthesizable as FSMs. Finally we perform a case study, adding assertions to a cache-controller, and investigate the impact of synthesized assertions on circuit timing and area.

2 Related Work

Synthesis of assertions into traditional RTL code is currently available in IBM's FOCs tool, which translates PSL

assertions into VHDL or Verilog [7]. The language Lava [4] represents properties as circuits, but generally uses external formal-verification tools to prove that the outputs of these properties always hold, although it is capable of simulating them. The Vis tool [6] integrates synthesis and verification, but checks assertions using formal verification rather than synthesizing assertions as hardware. Oliveira and Hu introduce a high-level generate circuits from assertions [11], but these assertions may only be monitors at interface boundaries.

SystemVerilog Assertions were developed as part of the SystemVerilog standard and are documented in the Language Reference Manual [2]. Havlicek et al have meticulously developed SVA semantics, particularly as they relate to local variables [5]. We build upon their work in Section 5.

3 SystemVerilog Assertions

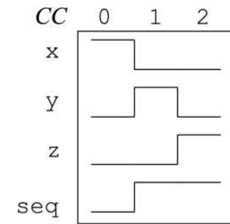
SystemVerilog Assertions were developed as part of the SystemVerilog language standard. They are semantically aligned with PSL, yet have Verilog-like syntax. SVA also adds certain new features not found in PSL.

Sequences are the building blocks of temporal assertions in SVA. Sequences are boolean expressions over multiple clock cycles. For example:

```
sequence reqack;
      req && data_in == 0
   ##1 data_in > 0 [*3:5]
   ##1 ack && data_in == 0;
endsequence
```

This sequence states that on the first clock cycle `req` must be high (true) and `data_in` low. The `##1` means that on the next cycle `data_in` must be greater than zero. It must remain greater than zero for three-to-five clock cycles. Finally `ack` must be true and `data_in` must return to zero. Semantically, on each clock cycle a sequence can either match, or not match. A sequence can generate multiple matches over the span of many clock cycles, as demonstrated in Figure 1. Note that simple boolean expressions are trivially sequences of length 1.

SVA sequences are used to build *properties*, which are expressions of expected behavior. Properties often include the implication operator:




```
sequence seq;
  (x ##1 y) or (x ##1 y ##1 z);
endsequence
```

Figure 1: The above sequence matches both on CC 1 and CC 2

sequence `|->` *property*
sequence `|=>` *property*

The *s* `|->` *p* operator states "if sequence *s* matches, then property *p* must hold." The `|=>` operator is similar, but begins checking *p* on the subsequent clock cycle. For example, the property "if there's a `req` and then `data_in` is greater than zero then `fifo_in` must not be full" could be expressed as follows:

```
property goodbuffer;
    (req ##1 data_in > 0) |->
        !fifo_in.full;
endproperty
```

By definition all finite sequences count as properties. Such converted sequences have the `first_match` operator applied. This means that if the sequence matches at least once then the property holds, whereas if the end of the sequence is reached without a match then the property does not hold.

Properties are checked via *assertion* statements:

```
always assert property (goodbuffer);
```

This says that on every clock cycle property `goodbuffer` must hold.

In this paper we will deal with assertions that must always hold. The techniques presented are extendible to assertions that are checked only at specific points in time.

Additionally, assertions dealing with multiple clock domains are beyond the scope of this paper. Finally, we do not examine SVA local variables, but the techniques presented in this paper are applicable to them with little extension.

4 Bluespec SystemVerilog

Previous research has developed Bluespec, a system for high-level synthesis from guarded atomic actions [3, 9]. Bluespec SystemVerilog (BSV) is a recently developed tool which implements the Bluespec semantic model in SystemVerilog. Additionally, BSV adds support for higher-order functions, polymorphic types, and strong static elaboration capabilities. BSV replaces certain Verilog constructs, such as `always` blocks, with rules, a higher-level way to express concurrent behavior.

4.1 Bluespec Semantics

In the Bluespec semantic model, all hardware state is explicitly defined. The designer then specifies operations to be performed on state elements through *rules*. Here is a simple rule for a cache controller in BSV. This rule iterates through all cache locations, and writes back to memory all locations that are dirty:

```
// Write back all contents of the cache
rule sync_cache(state == Synchronize);
  case (cache[index]) matches
    tagged Valid {.tag, .data, .isDirty}:
      if (isDirty) begin
        writeToMemory({index, tag}, data);
        notDirty(index);
      end
    default:
      noAction;
  endcase
  state <= (index == ‘MAX_ADDRESS)?
             Ready :  Synchronize;
  index <= index + 1;
endrule
```

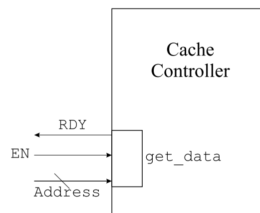
Rules are guarded atomic actions. This has two important consequences. First, the rule represents an atomic transaction — either all the actions that the rule describes will happen, or none of them will. Second, the action is guarded by a predicate. In the above example, the rule `sync_cache` may only “fire” when the cache controller state is `Synchronize`. When every cache line has been written back the controller will proceed to the `Ready` state and this rule will stop executing. Contention between resources is automatically managed by the Bluespec scheduler, described in Section 4.2.

Rules represent atomic actions that trigger on internal conditions. Bluespec *methods*, on the other hand, are triggered by external modules. For example, this is a cache controller method for retrieving an element from the cache:

```
method Action get_data(Address addr)
                 if (state == Ready);
  Index i = get_index(addr);
  case (cache[i]) matches
    tagged Invalid:
      getFromMemory(addr);  //cold miss
    tagged Valid {.tag, .data, .isDirty}:
      if (tag == get_tag(addr))
        sendToProc(addr, data);  //hit
      else  //conflict miss
        getFromMemory(addr);
  endcase
endmethod
```

This method, given an address, attempts to retrieve that address from a direct-mapped cache. If the address is present, it returns it to the processor. Otherwise it is retrieved from main memory.

Methods are translated into module ports as shown:



The Bluespec compiler enforces that methods may only be called if the `RDY` signal is true. This ready signal represents the method’s implicit condition. In the above example, the `get_data` method may only be called if the controller is in the `Ready` state. Methods integrate seamlessly into rule semantics because implicit conditions become part of the calling rule’s condition, and the method’s

actions, if any, become part of the rule's atomic transaction. In Bluespec an `Action` method causes a change in state, a `Value` method simply returns a value, and an `ActionValue` method does both atomically.

Module interfaces in BSV are collections of methods. Here is an example interface for a cache controller:

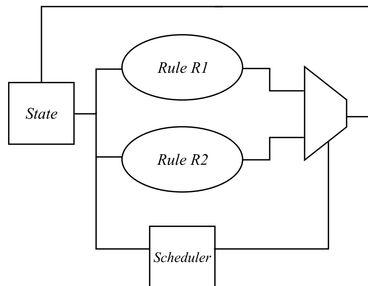
```
interface CacheController;
  method Action get_data(Address addr);
  method Action write_data(Address addr,
                                  Value v);
  method Action sync();
  method Action flush();
endinterface
```

Different modules which offer a particular interface are interchangeable. Thus the designer may write a direct-mapped cache and use it for simulation and verification, then perform architectural exploration on various caching schemes, all without changing the surrounding control logic.

4.2 Rule Scheduling

One strength of the Bluespec semantic model is that the designer can reason about each rule in isolation, as if no other rule is acting on state simultaneously. However, actually firing one rule per clock cycle will not result in good performance. Thus the Bluespec compiler uses a scheduler to ensure numerous rules may fire on the same clock cycle.

It is easy to see that if rules R1 and R2 read and write disjoint state elements they may safely operate in parallel. When they affect overlapping state the scheduler must make a decision as to which rule should fire. This results in the following hardware:



For an in-depth discussion of scheduling considerations see [12]. A requirement for our synthesizable assertions is that their presence cannot change the schedule of the module they are testing.

4.3 Assertions in BSV

One advantage of assertion-based verification is that it makes a designer's implicit assumptions explicit. Given that Bluespec's semantic model already makes many assumptions explicit (See Section 6.4), what kinds of assertions will a designer want to add? Most common are *functional* assertions. These assertions serve as an additional check that the device-under-test is behaving according to specification. Also useful are *performance* assertions. These check that the device is performing its task with sufficient throughput and latency. Finally, the designer can use *statistic-gathering* assertions. Rather than simply passing or failing, these assertions gather information on the device under test.

5 SVA Semantics in BSV

The main question is how to represent SystemVerilog Assertions, which are clock-cycle based, in BSV semantics, which have no notion of clock cycles until the rules are scheduled. Our approach is to transform the assertions themselves into Bluespec modules. Thus our synthesized assertions will be subject to the same semantic model and go through the same scheduling process as the devices they are testing.

One consequence of this is that the user is able to make synchronous, clock-cycle based assertions even though BSV rule semantics are without clocks. Thus an assertion in BSV is asserting that a property must hold across all possible schedules.

For example, suppose the designer writes an assertion "A `req` will be followed by an `ack` in at most three clock cycles." If the scheduler chooses a schedule where this property does not hold, then the resulting hardware will be in error. The presence of the assertion allows the designer to locate the problem and correct the schedule. In the future this could be extended to control the scheduler directly through synchronous extensions to the semantic model, as in [10].

5.1 SVA to FSM Translation

Temporal logic assertions are naturally mapped into Finite State Machines. To support this we have created a library of reusable FSM modules representing primitive temporal logic building blocks. High-level assertions are translated into basic temporal logic using the methods outlined in Appendix H of the SystemVerilog LRM [2]. Our library contains a synthesizable module for each primitive operator written in BSV itself. The Bluespec compiler instantiates the appropriate modules and compiles them using the normal compilation path.

The complete list of SVA primitive operators is given in Figure 2. Each production in the grammar corresponds to a synthesizable BSV module in our library. By using BSV's parametric polymorphism and first-class objects we are able to make the assertion modules both general and succint. The entire library was implemented using 513 lines of Bluespec code. Having a source-code level library available eases maintenance and improvement.

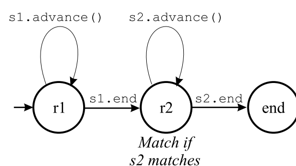
5.2 Sequence Translation

We begin by defining a Bluespec interface to represent sequences:

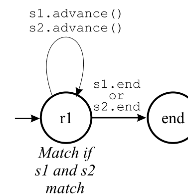
```
interface Sequence;
  method ActionValue#(Bool) advance();
  method Bool ended();
endinterface
```

This interface has two methods. The first advances the state of the sequence FSM, and returns a boolean value that indicates whether or not the sequence matches on this clock cycle. The second is a pure value method that returns true when the sequence has ended. By convention we will ensure that when all our FSMs halt they return to their initial state. Thus they can be immediately reused if necessary, as we will see below.

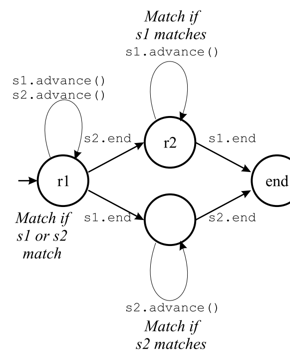
For example, the concatenation operator `##1` first runs sub-sequence *s1* to completion, then on the next clock cycle begins to run *s2*:



The combined sequence matches only when *s2* matches. Applying the `intersect` operator to two sequences means that the combined sequence matches on a clock tick if and only if both sub-sequences *s1* and *s2* match:

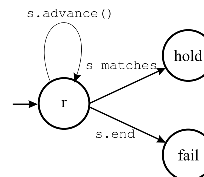


This means that if either sequence ends the combined sequence can end as it cannot result in any more matches. This is in contrast to the `or` operator, which may match whenever either subsequence matches, and thus must continue to run until both sub-sequences have halted:



5.3 Property Translation

In contrast to sequences, properties only result in one final result — either they hold or they do not. As such they are straightforward to represent as FSMs. The following FSM corresponds to the "sequence form" line of Figure 2. It translates a sequence into a property by holding and ending the first time the sequence matches:



```
//Sequences
R ::= b                    // "boolean expression" form
      | ( 1, v = e )       // "local variable sampling" form
      | (R)                // "parenthesis" form
      | ( R ##1 R )        // "concatenation" form
      | ( R ##0 R )        // "fusion" form
      | ( R or R )         // "or" form
      | ( R intersect R )  // "intersect" form
      | first_match ( R )  // "first match" form
      | R [ *0 ]           // "null repetition" form
      | R [ *1:$ ]         // "unbounded repetition" form

//Properties
P ::= R                      // "sequence" form
      |(P)                   // "parenthesis" form
      | not P                // "negation" form
      | ( P or P )           // "or" form
      | ( P and P )          // "and" form
      | ( R |-> P )          // "implication" form
      | disable iff ( b ) P  // "reset" form
```

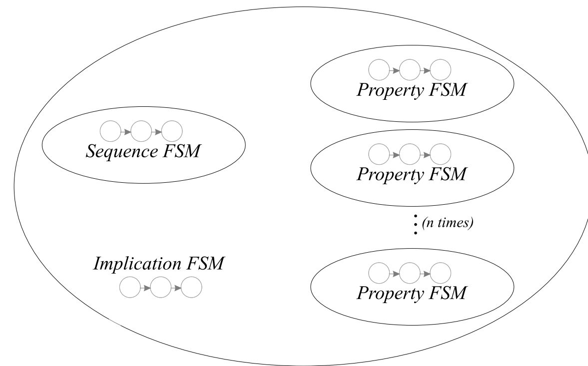
Figure 2: SVA primitive temporal logic operators (Source: SystemVerilog LRM [2])

The implication operator `|->` is the most challenging primitive to implement. Recall that this operator takes a sequence as its antecedent and a property as its consequent. Every time the sequence matches the property must hold. Note that both the running of the sequence and the property may span multiple clock cycles, and that the sequence may match multiple times.

To implement this we require more than one copy of the property FSM. Every time the sequence FSM matches we must start a new property FSM. After the sequence ends, if all property FSMs hold, then the entire operation holds. Thus we must disallow infinite sequences in the implication antecedent as these would never terminate and thus never result in the property holding.

This is where the capabilities of synthesized assertions deviate from software-simulation checking. In software we may continue to spawn new property-checking threads every clock cycle until the simulator terminates. In hardware, in contrast, we are limited to a fixed number of circuits.

However, we can still represent the implication operator using one sequence FSM and multiple property FSMs. In the worst case the sequence FSM will match on every clock cycle. Therefore the implication machine will contain a number of identical property FSMs equal to the longest path in that FSM:



For example, in the following implication:

```
s |-> ((x ##1 y ##1 z) or (a ## b));
```

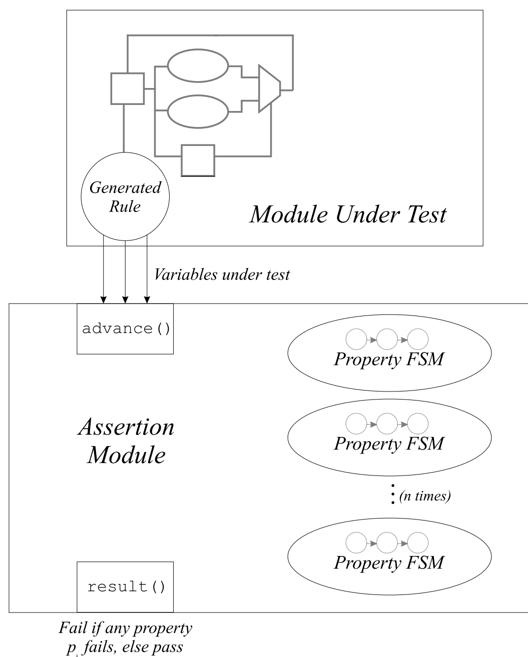
the consequent has a worst-case length of 3 clock cycles. Thus having three copies of that FSM will ensure that the implication automaton will always have at least one property FSM ready in the initial state, even if the sequence *s* matches on every clock cycle.

The SVA standard allows for recursive property definitions. However, we must exclude such descriptions from the SVA subset we map to hardware, as they do not result in a compile-time enumerable longest path. As an optimization, if we can prove at compile time that the se-

quence FSM will generate at most n matches, then we need at most n copies of the property FSM.

5.4 Assertion Translation

Checking assert statements themselves is very similar to checking implication. To check that a property always holds, we must begin a new property FSM on every clock tick. In order to ensure that we will always have an FSM in the initial state, we must have a number of identical modules equal to the longest path in that FSM, exactly as above.



Note that the compiler has added a rule to the module-under-test. All of our FSMs communicate with Bluespec methods. But all chains of method calls can eventually be traced to a rule. Thus when synthesized assertions are included in a module, the Bluespec compiler adds a single rule which reads the values of all variables under test, and advances the assertion FSMs. Because the assertion module only reads values from the module-under-test, it does not destructively interfere with the scheduling of that module's rules.

5.5 Language coverage

The concept of a hardware description language's "synthesizable subset" is by no means a new one to hardware designers. Using the method of synthesis outlined in this paper we are able to support a significant percentage of SVA features, as shown in Figure 3.

6 Case Study: Cache Controller

To investigate the use of synthesizable assertions in Bluespec we performed a case study using a cache controller. The cache controller manages a two-way set-associative cache with a write-through, allocate-on-write policy. The controller's design is outlined in Figure 4. It takes loads from the CPU and requests the appropriate line from both cache-ways, then enqueues the request in a FIFO. In the following clock cycle the cache responses are compared versus the actual address. On a hit the value is immediately returned to the processor and the FIFO dequeued. On a miss the request is sent to memory. When the memory returns the value is simultaneously cached and returned to the processor, and the FIFO dequeued.

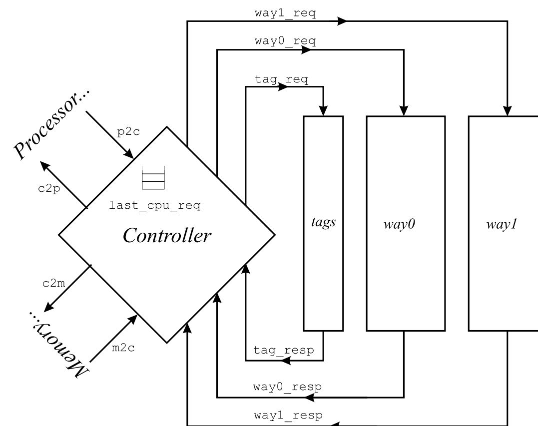


Figure 4: Cache Controller design overview

6.1 Performance Assertions

First we came up with specifications for performance assertions. An example specification is:

Supported:	Unsupported:

Sequence Operators:
`##n`, `[* ]`, `[= ]`, `[-> ]`
`throughout`, `within`
`intersect`, `and`, `or`
`$` (unbounded repetition)
Property Operators:
`not`, `or`, `and`, `|->`, `|=>`, `if...else`
`assert` Statements
	Unbounded repetition in implication antecedent
	Recursive properties
	`expect` statements
	`cover` statements
	`disable iff` operator

Figure 3: Summary of language feature support

> When a cpu request is made a cache memory read is made in the same cycle. For write requests, main memory and cache memory are written in next cycle. For read requests, either main memory is read or result returned in next cycle.

We translated these specifications into SVA properties. The above specification was translated into two separate properties:

```
property cpu_read_perf;
  read_request |->
        isRead(way0_req)
     && isRead(way1_req)
     && isRead(tag_req)
     ##1 isRead(c2memory_req)
         || isRead(c2p_data);
endproperty

property cpu_write_perf;
  write_request_rw |->
        isRead(way0_req)
     && isRead(way1_req)
     && isRead(tag_req)
     ##1 isWrite(c2memory_req)
         && isWrite(tag_req)
         && (isWrite(way0_req)
         || isWrite(way1_req));
endproperty
```

6.2 Functional Assertions

Next we designed specifications for functional assertions. An example functional assertion is "On a write request only one cache-way is written." This translates into an SVA property as follows:

```
property goodWriteRequest;
  write_request |=>
    if (cache_tag_resp.next_evict_way0)
         isWrite(way0_req)
      && !isWrite(way1_req)
    else isWrite(way1_req)
      && !isWrite(way0_req);
endproperty
```

One functional property we wished to check is "No address can be cached in both way0 and way1." This could be implemented as an invariant over all cache tags. However this would result in a large amount of combinational logic. An alternative strategy is to check that this property holds for every line that the cache reads. This is a weaker invariant, but results in much more efficient hardware. The result was the following property:

```
property no_double_caching;
  read_request_rw |=>
    tagsDoNotMatch(cache_tag_resp);
endproperty
```

6.3 Assertions and Implicit Conditions

As explained in Section 4.1, Bluespec methods can have implicit conditions on their use. For example, a Bluespec FIFO ensures that the FIFO may not be dequeued if it is empty, or enqueued if it is full. In some cases these conditions alleviate the need for certain assertions that would

normally need to be written explicitly in SVA. For example, when exploring ideas for functional assertions for the cache controller, we came up with the following:

> Whenever a response from memory is received, the cache controller state must be `Waiting_For_Memory`.

This assertion proved to be unnecessary, as it was already enforced by an implicit condition on the `m2c` method:

```
method Action m2c(mem_resp)
  if (state == Waiting_For_Memory
    && isRead(last_cpu_req.first()));
  ...
```

This is enforced by the Bluespec scheduler as a precondition of using the method, so an explicit assertion is not needed.

However, there are times when implicit conditions work against assertions. SVA assertions are synchronous, and thus must act every clock cycle. However if the user attempts to make assertions about methods with implicit conditions, then these conditions will be transferred to the assertion itself, and it may not be able to act.

To resolve this we have decided that assertions may not refer to values with implicit conditions. This is enforced via Bluespec's existing `no_implicit_conditions` attribute. As future work assertion support could be expanded to deal with implicit conditions directly.

6.4 Statistic-Gathering Assertions

In addition to functional and performance assertions, we used synthesizable assertions to gather statistical data on the cache performance. SVA allows the user to add pass statements and fail statements that are executed when the assertion does/does not hold. We use this to add counters for statistics such as total cache read requests, read hits, and total memory requests. A typical statistic gathering assertion is as follows:

```
property count_read_hits;
  read_request |=>
    isValid(c2p_data);
endproperty

always assert property (count_read_hits)
  read_hits <= read_hits + 1;
 else
  read_misses <= read_misses + 1;
```

In addition we added a mechanism for tracking average read response time. To do so we added a counter to track `outstanding_requests` and `total_response_cycles`. Every clock cycle `total_response_cycles` is incremented by the number of `outstanding_requests`. We can calculate average response time simply by dividing `total_response_cycles` by `total_read_requests`. This calculation could be done in hardware with a divider module, but this would have a significant impact on circuit area and timing. Therefore we simply add the above registers and perform the calculation externally.

It should be noted that all of these statistic-gathering functions could have been implemented with standard Bluespec rules. However there are some benefits to using assertions. First of all, they are easily disabled with compiler directives. Secondly, they may include an **`else`** clause, as seen above. Most importantly, they can use SVA temporal logic features to implicitly define finite state machines. A potential direction for future work is expanding Bluespec rule predicates to use temporal logic.

7 Synthesis Results

Three versions of the cache controller were synthesized: the original baseline version with no assertions, a version with only functional and performance assertions, and a version with all assertions including the statistic-gathering counters. The designs were first synthesized using Magma Mantle version 4.1.18 with a 0.13μm library and a clock period constraint of 1.0 ns.

The results are shown in Table 1. The simple performance and functional assertions increased circuit area by 9%, whereas the statistic-gathering counters increased area by 104%. Investigation showed that this latter increase is primarily due to the eight 32-bit registers that this version of the design uses as counters, rather than the assertions which drive them. We then experimented with timing constraints of 2, 1.5, and 0.75 ns. The results are shown in Table 5. The statistic-gathering design was unable to meet the .75 ns constraint. The baseline design was not synthesized at 2 ns.

Design:	Circuit Area (μm^2)	Approx. Gates	Change
Baseline Cache Controller	26771	~5354	-
Controller with functional assertions	29073	~5815	+9%
Controller with statistic-gathering counters	54665	~10933	+104%

Table 1: Synthesis results for 1.0 ns timing constraint

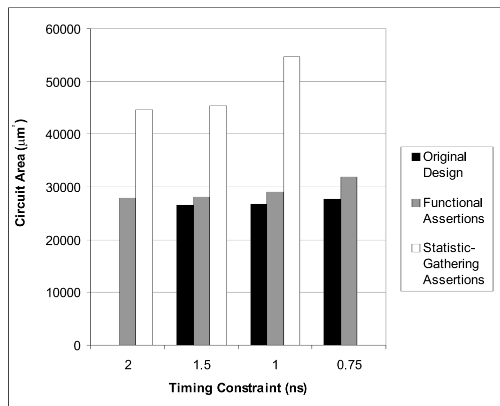


Figure 5: Synthesis Results for various timing constraints

8 Conclusions

Bluespec SystemVerilog uses an unclocked semantic model, but we have demonstrated that there are situations when the designer wishes to make synchronous assertions. We have shown that by translating assertions into Bluespec modules themselves we can craft synchronous assertions even though Bluespec's semantic model is unclocked. Additionally we have demonstrated that a significant subset of the SystemVerilog Assertions language can be synthesized using this method. Finally we have shown that assertion hardware can be included in a design without significant overhead in terms of combinational logic.

This research has uncovered many potential future research directions, including support for making assertions about the Bluespec semantics itself, using SVA temporal logic for Bluespec rule predicates, and ultimately using the assertions to control the scheduling and compilation process directly by extending the work in [10].

Synthesizable assertions have opened up new possibilities for the designer. They can serve as a convenient temporal logic language to describe finite state machines. They can be used for accelerated verification using FPGAs. They can even remain in final hardware for dynamic verification and fault-tolerance. We believe that synthesizable assertions are a useful tool to add to the designer's toolbox.

References

[1] Accellera. *Property Specification Language Reference Manual*, 2004. www.eda.org/vfv/docs/PSL-v1.1.pdf.

[2] Accellera. *SystemVerilog 3.1a Language Reference Manual*, 2004. www.eda.org/sv/SystemVerilog_3.1a.pdf.

[3] Arvind, Rishiyur S. Nikhil, Daniel L. Rosenband, and Nirav Dave. High-level Synthesis: An Essential Ingredient for Designing Complex ASICs. In *Proceedings of ICCAD'04*, San Diego, CA, 2004.

[4] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in haskell. In *International Conference on Functional Programming*, pages 174–184, 1998.

[5] John Havlicek et al. *Notes on the Semantics of Local Variables in Accellera SystemVerilog 3.1 Concurrent Assertions*, 2004. www.accellera.org/activities/techrep/techrep.pdf.

[6] R. K. Brayton et al. Vis: A system for verification and synthesis. In *Eigth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.

[7] Yael Abarbanel et al. FoCs: Automatic generation of simulation checkers from formal specifications. In *Computer Aided Verification*, pages 538–542, 2000.

[8] Harry D. Foster, Adam C. Krolnik, and David J. Lacey. *Assertion-Based Design, Second Edition*. Springer, Boston, MA, 2004.

[9] James C. Hoe and Arvind. Synthesis of Operation-Centric Hardware Descriptions. In *Proceedings of ICCAD'00*, pages 511–518, San Jose, CA, 2000.

[10] Grace Nordin and James C. Hoe. Synchronous Extensions to Operation-Centric Hardware Description Languages. In *Proceedings of MEMOCODE'04*, San Diego, CA, 2004.

[11] Marcio Oliveira and Alan Hu. High-level specification and automatic generation of ip interface monitors. In *Proceedings of the Conference on Design Automation (DAC '02)*, pages 129–134, 2002.

[12] Daniel L. Rosenband and Arvind. Modular Scheduling of Guarded Atomic Actions. In *Proceedings of DAC'04*, San Diego, CA, 2004.