Bluespec SystemVerilog Known Problems and Solutions

Revision: 24 November 2008

Contents

Introduction

This document records known issues in the Bluespec SystemVerilog compiler and how they can be worked around.

Problem #001: Mutually-exclusive uses of port-limited resources may be allocated inefficiently

Example

Assume a ROM that supplies a single-ported read method that is used in the following rule (to assign registers `x` and `y`):

```
rule broken (True);
  if (p)
    x <= rom.read(addr_a);
  else
    y <= rom.read(addr_b);
endrule
```

The compiler will not deduce that uses of the read method in separate if branches are mutually exclusive and compilation will fail with a resource allocation error. It is also possible (when mutually exclusive uses are in separate rules) for inefficient schedules to be generated, though this is less likely.

Solution

Replace the the mutually exclusive ROM uses with a single, conditionally controlled use as follows (assuming the output of the read method has type `Bit#(16)`):

```
rule working (True);
  Bit#(16) tmp = p ? rom.read(addr_a) : rom.read(addr_b);
  if(p)
    x <= tmp;
  else
    y <= tmp;
endrule
```

Common subexpression elimination should normally handle any repeated predicates (though it may make the code simpler to factor that out into a local assignment as well).

Problem #002: Changing compiler switches does not force recompilation when required

Example

When using the "-u" flag, the compiler only checks whether a ".bo" file exists for the given ".bsv" file. If the object exists and has a date which is more recent than the date of the source file, then no recompilation is performed.

However, the source file is not the only source of input for generating an object file. The command-line compiler flags also impact the generation. Recompilation should check not only whether the source file has changed, but whether command-line generation flags have changed. Currently, this check is not performed.

This command generates `sysFoo.o` from `Foo.bsv`:

```
% bsc -sim -g sysFoo Foo.bsv
```

This recompilation command ought to generate `sysFoo.v` from `Foo.bsv`, but it does nothing, because the source file has an earlier date than the object file:

```
% bsc -u -verilog -g sysFoo Foo.bsv
```

Solution

Removing the "-u" flag will force compilation regardless of the state of any existing files. However, it won't recompile any imported files.

Removing all the compiler-generated intermediate files will force recompilation of everything:

```
% rm *.bi *.bo
% bsc -u ...
```

Problem #005: Methods are forced to be more urgent than rules

The scheduler considers methods to be the most urgent things in a module and allows them to block rules whenever they are enabled (as part of implementing the standard interface contract - if a method that is ready is enabled, it will be executed). Sometimes it is desirable to provide a method that is less urgent than an internal rule.

Solution

Change the ready signal of the method so that it does not overlap with the enable condition of the desired more urgent rule. This can usually be achieved by anding `!p` (where `p` is the explicit condition of the relevant rule) with the rest of the method's ready signal.

Problem #006: Bit selection out of range (with a large positive index)

In certain cases, the compiler ignores the sign of constants used in array and bit extraction. This means that out-of-range errors related to negative constants can appear as out-of-range errors involving large positive constants (the twos-complement representation of the negative number).

Solution

When an out-of-range error involving a large positive constant appears (particularly constants that are close to 2^{32}), look for negative arguments to bit or array extraction (which might be caused by negative literals or loops that execute beyond their safe boundaries).

One common way for loops to execute beyond their safe boundaries is if their termination condition involves anything dynamic (e.g. a value read out of a register or a method parameter). This is because the compiler must be able to statically evaluate a termination condition to True before it can stop elaborating a loop. When a loop termination condition only involves static (i.e. compile-time) values this is not a problem. When the termination condition involves a dynamic value, however, the compiler must prove that the termination condition is irrelevant to the final result in order to stop unrolling the loop (which is a more significant obligation that cannot always be met when it is expected).

Problem #007: Generation of C or Verilog from a Bluespec module hangs

This can be caused by a number of issues (including):

- an infinite loop in statically elaborated code (when the `-steps` options is used)
- an explosion in the number of rules that slows down the scheduler
- complex predicates which take a long time in disjointness testing

Solution

If a code generation is taking a long time and you suspect the compiler is hanging, the first thing to do is to rerun the compiler with the `-v` and `-show-stats` options to see what phases are taking a long time in the compiler and statistics about intermediate data (e.g. the number of rules and internal definitions.

If the compiler is taking a long time in the "typecheck" stage, this is probably being caused by certain constructs (such as large, complex case statements) that are slow to typecheck.

If the number of rules or internal definitions is exploding that is the likely cause of the problem. Large numbers of rules can be caused by generation constructs that create rules or by use of the `-expand-if` flag when there are a large number independent if statements in a single rule. The solution in this case is to change your program and/or your use of the `-expand-if` flag so the number of rules does not explode.

If the number of internal definitions is exploding this can also be caused by the default inlining of combinatorial functions. Complex combinatorial function (particularly functions that consist of large case statements) can create many local definition if they are inlined and not simplified. In that case, one possible solution is to mark any complex combinatorial functions with the "`(* noinline *)`" attribute so they are compiled separately.

If the compiler is taking a long time in the scheduling phase and the number of rules is not large, the compiler is probably taking a long time in disjointness testing. The easiest way to confirm this is to rerun the compile with the `-scheduler-effort` flag to a low value (or 0 to disable disjointness testing altogether). If the compile now gets through the scheduling phase, then the problem is in disjointness testing. Making the predicates of your rules simpler might help that.

Problem #008: Compiler expects the proviso `Add#(a,b,TAdd#(a,b))` or `Mul#(a,b,TMul#(a,b))`

Example

The compiler may infer that the proviso `Add#(a,b,c)` is needed (or similarly for `Mul`). If types `a` and `b` are known (either because they are specific types or because they are variables bound in the base type) but the type `c` is not known, then the compiler should know how to add the known types together to determine the unknown, so the proviso should not be required. However, instead of discarding the proviso, the compiler actually reports that `Add#(a,b,TAdd#(a,b))` is needed (or similar for `Mul`).

Solution

Make the compiler happy by adding a proviso. Either add `Add#(a,b,TAdd#(a,b))`, or add `Add#(a,b,c)` where variable `c` is a fresh variable which is not named in the base type.

Problem #009: Type-checking large case statements can take forever

Symptoms

If the compiler takes a long time in the typechecking phase, it may be due to a case statement with many clauses (32+) that introduces provisos in each arm. Due to limitations in the way that case statements are type checked, a case statement with 32 such clauses may take a minute or a few minutes. A case statement with 64 such clauses may take several minutes to forever.

Workarounds

If you encounter this problem, try reducing the number of introduced provisos or reducing the number of clauses in the case-statement. If the case-statement is implementing an algorithmic function or procedure, try rewriting it as a function.

Provisos are typically introduced by using numbers (`0`, `1'b0`, etc), using bit-concatenation, using array indexing (`arr[0]`), and by calling functions which carry provisos. These provisos are typically resolved by the context (e.g., the size of the number `0` is determined by the signal which is assigned that value). Type checking of case statements is hindered if the proviso is not satisfied inside the individual clauses. Thus, with many clauses, a large set of provisos is generated.

One way to resolve provisos inside clauses is to add more explicit types. These act as firewalls for the typechecker by fixing down the types of values. For example, in this contrived example:

```
case (x)
  4b0000:  begin
             x = {y,z};
           end
endcase
```

the size of the concatenation can be fixed by introducing a type:

```
case (x)
  4b0000:  begin
             Bit#(8) tmp = {y,z};
             x = tmp;
           end
endcase
```

Another way to remove provisos from the case arms is to declare variables of fixed size outside of the case statement. For example, this code:

```
case (x)
  4b0000:  begin
             x = f(arr[0]);
           end
endcase
```

can be rewritten as:

```
Bit#(8) val0 = f(arr[0]);
case (x)
  4b0000:  begin
             x = val0;
           end
endcase
```

Problem #011: Proviso errors when deriving a class for types whose subtypes do not have instances of that class

Example

The following example fails becase it attempts to derive the `Bits` class for a type whose subtypes (here `Integer`) are not members of the `Bits` class:

```
typedef struct { Integer x; Integer y; } Coord deriving(Bits);

module mkTest (Empty);
  Reg#(Coord) c();
  mkRegU c_reg(c);
endmodule
```

The error message that results is:

```
"Test.bsv", line 3, column 8: (T0031) Error:
  The provisos for this expression could not be resolved because there are no
  instances of the form:
    Prelude::Bits#(Test::Coord, _tctyvar1007)
  The proviso was implied by expressions at the following positions:
    "Test.bsv", line 5, column 3
```

Solution

If possible, add deriving to the subtype or define an instance of the class for the subtype.

This may not always be possible, however. The `Integer` type can never have an instance of the `Bits` class. Thus, any data type which contains an `Integer` cannot have `Bits` derived for it.

Problem #014: Typechecking hangs

If typechecking hangs, it may be due to an infinite loop in the user's code which the compiler currently does not detect.

The `SizeOf` type function can be used inside a type to return the size of a type in bits. This function only works if a bit-vector interpretation for the type exists. The function finds the size by looking for an instance of the `Bits` class for the given type.

Thus, if the user calls `SizeOf` while defining an instance of `Bits` for that type, he has created a circular dependency. The compiler will loop forever trying to determine the size.

Solution

Don't use `SizeOf(t)` inside the `Bits` definition for type `t`.

Problem #017: Module instantiation required immediately after interface instantiation

Symptoms

Code which instantiates a module separately from its interface instantiation causes errors; e.g.,

```
module foo();
  Reg#(Bool) r_ifc();
  int x;
  mkRegU r_inst(r_ifc);
  ...
endmodule
```

The known error caused by this is a parse error (P0005), reported at the declaration following the interface (e.g., above, "int x;").

Workarounds

Instantiate modules immediately after their interfaces; e.g., rewrite the code above as

```
module foo();
  Reg#(Bool) r_ifc();
  mkRegU r_inst(r_ifc);
  int x;
  ...
endmodule
```

Problem #020: Type error when applying unary bitwise operators on non-bit type mentions strange function name

Example

The unary bitwise operators `&`, `|`, `^`, and their negated forms with `!` are represented in the compiler as functions with names `reduceAnd`, `reduceOr`, `reduceXor`, `reduceNand`, `reduceNor`, and `reduceXnor`.

If you observe a type error that refers to this name, it is referring to an incorrect application of the associated unary operator.

For example, this message:

```
"Test.bsv", line 9, column 12: (T0020) Error:
  Type error at:
  Prelude::reduceOr

  Expected type:
  function Prelude::Bool f(c x1)

  Inferred type:
  function a#(1) f(a#(b) x1)
```

is informing the user that the operator `|` is being being used to produce a Boolean value, when in fact it returns a single bit.

Problem #021: Wrong number of arguments to a type constructor manifests as unbound type constructor error

If an enumerated type or union tag is applied to two or more arguments, an error will be reported about an unbound type constructor. The constructor name will be the type name followed by the tag name, with a non-ASCII center-dot character separating the two. The position of the error correctly points to the location where the constructor is applied to too many arguments.

Example

The following code:

```
typedef enum { Red, Blue, Green } Color;
Color c1 = Red;
Color c2 = Red(True,False);
```

produces this error:

```
"EnumTwo.bsv", line 5, column 12: (T0007) Error:
  Unbound type constructor ‘EnumOne.Color\uffffRed’
```

Problem #023: Sub-vector assignment may take a long time to synthesize

Example

The following code:

```
Reg#(bit[511:0]) r <- mkRegU;

...

r[255:0] <= 256'hdeadbeef;
```

takes a long time to synthesize. BSC eventually stops, producing an error message similar to

```
"Prelude.bs", line 794, column 0: (G0024) Error:
  The number of function unfolding steps has been exceeded when unfolding
  ‘Prelude.map1’. Use flag ‘‘-steps N’’ to set the number of steps. Current
  value is 50000.
```

This applies to assignments to bit vectors as well as to vectors of other types, but is insignificant when the size of the vector (512 in the example above) is small.

Solution

Employ concatenation instead of sub-vector updates:

```
r <= {r[511:256], 256'hdeadbeef};
```

Problem #024: VCD differences between Bluesim and Verilog simulation

Bluesim is a Bluespec-level cycle-based simulator that is architecturally different than an event-based RTL-level simulator. Some of these differences lead differences in VCD output when simulating the same design. Architectural differences that lead to different VCD output include:

- Bluesim transfers no information when a method is *not* called. This means the argument values dumped by Bluesim for method calls may not match the argument values dumped by a Verilog simulation. In particular, no value changes will be dumped for the arguments of an `Action` or `ActionValue` method, when the enable signal for that method is 0. For the same reason, the data bits of an RWire output (and any values or value changes that depend on those data bits) may differ between Bluesim and Verilog simulations for any cycle where that RWire's output is not valid.
- Bluesim does not evaluate local temporaries that are not required to compute the next cycle's state. This means Bluesim will not dump value changes for such local temporaries, even though the value of those local temporaries might have changed (had they been computed).

Bluesim and Verilog VCDs and behavior also may differ when using unguarded (UG) FIFOs incorrectly. Since unguarded FIFOs do not enforce implicit conditions on the `enq` and `deq` methods, it is possible to `deq` from an empty FIFO or `enq` onto a full FIFO. If either of those events occurs, the subsequent behavior of a Bluesim or Verilog simulation is undefined (and the subsequent behavior, including dumped value changes, is unlikely to match). This is not an issue with standard Bluespec FIFOs because they enforce implicit conditions on `enq` and `deq`.

There are also some open bugs related to Bluesim VCD output:

- A bug in the Bluesim exit sequence can cause differences between Bluesim and Verilog VCD output at the end of a simulation. This includes Bluesim dumping value changes for times that are after the end of the simulation.

Workarounds

Work with the signals that will be the same between the two simulators. These include register values, registered state-element outputs, method enables and `CAN_FIRE` and `WILL_FIRE` signals.

Do not rely Bluesim VCD output in the last cycle of a simulation (and ignore any value changes dumped for times after the simulation ends).

Problem #025: C++ compiler bugs encountered while building a Bluesim model

Bluesim compilation generates C++ code for a BSV design and then uses the host system's installed C++ compiler to build the Bluesim executable. Bugs in the C++ compiler can trigger failures while building the Bluesim executable or incorrect simulation results while running the generated Bluesim model.

Bluespec recommends using a well-tested version of a mainstream C++ compiler, such as GNU C++, and avoiding beta releases and compilers with known problems.

Known C++ Compiler Bugs

- GNU g++ compilers in the 3.3.x series generate an incorrect instruction sequence for some code involving static inline functions when compiled with optimization level -O2 or greater. In rare instances, this can lead to incorrect simulation results with Bluesim. Bluespec recommends using GNU g++ 3.2.x or eariler, or upgrading to GNU g++ 3.4.6 or later to avoid this problem. If GNU g++ 3.3.x must be used, Bluespec recommends using the environment variable CXXFLAGS=-O1 to avoid triggering the bug in g++, though this may degrade Bluesim simulation performance.

Workarounds

When a bug in the C++ compiler is encountered, the preferred remedy is to upgrade the compiler to a version which does not exhibit the bug. If that is not possible for some reason, other techniques may be useful:

- If the bug is triggered by a particular optimization or feature, it can be disabled using the CXXFLAGS environment variable to pass options to the c++ compiler.

Problem #026: Missing or unhelpful positions in error messages for rules or methods with multiple clocks or resets.

BSC checks that each rule and method of a module is in a single clock domain. It is an illegal clock crossing (error G0007) when a rule or method contains expressions from different clock domains. Method calls on submodules are the only expressions which have associated clocks, so the error message lists the method calls of the rule (or method), grouped by clock domain.

Similarly, BSC checks that a rule or method has at most one associated reset. If it calls methods which are affected by multiple resets, then a warning is reported (G0043), and the method calls are listed, grouped by their reset.

The messages try to be helpful by printing a position for the method calls. Unfortunately, the positions are not always present or, when they are present, they point to the interface type declaration where the method was declared. This is not a helpful position for debugging the clock crossing problem.

Solution

Turn on the `-cross-info` flag when compiling to get more useful position information. The flag causes the positions to generally point to the place in the offending rule or method where the method

was called. Sometimes it defaults to pointing to the instantiation of the module which provides the method. In either case, these positions are an improvement.

With the `-cross-info` flag not enabled, the positions may not be helpful:

```
Error: "OCP_IFC.bsv", line 571, column 12: (G0007)
 Reference across clock domain in rule ‘slaveToMaster’.
 Method calls by clock domain:
   Clock domain 1:
     master.masterResp_putResponse at "OCP_IFC.bsv", line 248, column 23,
   Clock domain 2:
     slave.slaveResp_sThreadID at "OCP_IFC.bsv", line 342, column 52,
     slave.slaveResp_sData at "OCP_IFC.bsv", line 340, column 51,
     slave.slaveResp_sResp at "OCP_IFC.bsv", line 338, column 51,
```

By enabling the `-cross-info` flag, more meaningful positions may be provided:

```
Error: "OCP_IFC.bsv", line 571, column 12: (G0007)
 Reference across clock domain in rule ‘slaveToMaster’.
 Method calls by clock domain:
   Clock domain 1:
     master.masterResp_putResponse at "OCP_IFC.bsv", line 572, column 10,
   Clock domain 2:
     slave.slaveResp_sThreadID at "OCP_IFC.bsv", line 575, column 45,
     slave.slaveResp_sData at "OCP_IFC.bsv", line 574, column 45,
     slave.slaveResp_sResp at "OCP_IFC.bsv", line 71, column 6,
```

In some cases, no position is reported when `-cross-info` is not enabled:

```
Error: "Design.bsv", line 32, column 7: (G0007)
 Reference across clock domain in rule ‘always_fire’.
 Method calls by clock domain:
   Clock domain 1:
     t_out_data_reg.write
   Clock domain 2:
     t_in_data1.wget t_in_data1.whas
```

But the position is reported with `-cross-info` enabled:

```
Error: "Design.bsv", line 32, column 7: (G0007)
 Reference across clock domain in rule ‘always_fire’.
 Method calls by clock domain:
   Clock domain 1:
     t_out_data_reg.write at "Design.bsv", line 33, column 9,
   Clock domain 2:
     t_in_data1.wget at "Design.bsv", line 33, column 56,
     t_in_data1.whas at "Design.bsv", line 23, column 9,
```

Positions may not be reported with error G0043, when there is more than one associated reset and the `-cross-info` flag is not enabled:

```
Warning: "ResetCheckRule.bsv", line 8, column 6: (G0043)
 Multiple reset signals influence rule ‘test’.
```

```
 This can lead to inconsistent, non-atomic results when not all of these
 signals are asserted.
 Method calls by reset:
   Reset 1:
     r.read
   Reset 2:
     the_s.read
```

but with the `-cross-info` flag enabled more meaningful output is produced:

```
Warning: "ResetCheckRule.bsv", line 8, column 6: (G0043)
 Multiple reset signals influence rule ‘test’.
 This can lead to inconsistent, non-atomic results when not all of these
 signals are asserted.
 Method calls by reset:
   Reset 1:
     r.read at "ResetCheckRule.bsv", line 9, column 12,
   Reset 2:
     the_s.read at "ResetCheckRule.bsv", line 9, column 16,
```

Problem #027: Bluesim may produce a floating point exception in the case of a division-by zero

Bluesim may produce a floating-point exception in the case of a division-by-zero, even if the result of that division is not used in the computation.

For example:

```
if (i == 0)
      ...
else
      res <= 17 / i;
```

Can produce a floating-point exception when `i` is 0, even though no division result is being written to `res` in that case.

Workaround

The workaround for now is to use a redundant test to prevent the divisor from being 0. Example:

```
if (i == 0)
      ...
else
      res <= 17 / (i == 0 ? 1 : i);
```

This results in a division by 1 when $i == 0$. It prevents the exception in Bluesim and does not change the model behavior, since the result of the division is not used when $i == 0$.

Problem #028: C++ compilation can take an unreasonably long time using g++ version 4.2

In rare circumstances the C++ compilation stage of a Bluesim build can take an unreasonably long time using g++ version 4.2.

Workaround

A workaround is to specify C++ compilation flags that lower the optimization level. This can be done by setting the CXXFLAGS environment variable to:

```
-Wall -Wno-unused -O0 -g -D_FILE_OFFSET_BITS=64
```

This will result in a slower Bluesim simulation than when the normal optimization level is used.

Problem #029: Rule uses methods which have a rule that executes between them

Sometimes the user may encounter an error which looks like this:

```
Error: "Test.bsv", line 7, column 9: (G0096)
  The rule ‘RL_top_rule’ requires dynamic scheduling, which is not supported
  by Bluesim. This is because the rule uses methods which have a rule that
  executes between them in the static execution order of the separately
  synthesized submodule. See entry #29 in the KPNS document for more
  information and possible solutions.
  The methods and the rules between them are as follows:
    (sub.method1, sub.method2)
      RL_sub_rule
```

This occurs when a design is taking advantage of the fact that separately-synthesized modules have separate schedules. In an individual schedule, BSC insists that there be a static execution order of all rules and methods. By static, we mean that if rule A and rule B fire together in the same clock cycle, they will always fire in the same order.

When BSC generates separate Verilog modules, each module has its own schedule. There is no longer one static ordering of rules, but two static orders. This opens up the possibility that rule A from a parent module and rule B from a child module will execute in different orders in different clock cycles, because there is no global scheduler which is enforcing the order.

Consider the following rule in a parent module:

```
rule top_rule;
   if (p)
      sub.method1;
   else
      sub.method2;
endrule
```

If inside the submodule `sub`, there is a rule `sub_rule` which must execute after `method1` but must execute before `method2`, then the order of execution of that rule `sub_rule` and the parent rule `top_rule` will change, depending on the dynamic value `p`. In some clock cycles `top_rule` will execute first and in some clock cycles `sub_rule` will execute first.

Note that this feature only happens when the submodule is separately synthesized. If the synthesis boundary is removed, then `top_rule` and `sub_rule` will conflict, because a static execution order does not exist and BSC is forced to make the rules conflict.

This feature, of separately-synthesized modules, is supported by the Verilog which BSC generates. However, since the behavior changes when the synthesis boundary is removed, it is recommended that designers not rely on it.

Furthermore, Bluesim does not support this feature. So designs which are written this way cannot be simulated with Bluesim.

When a design like this is compiled for the Verilog backend, a warning is issued and Verilog is generated, but it is recommended that the user change his design. When the design is compiled for Bluesim, BSC exits with an error.

Solutions

Removing the synthesis boundary is one way to avoid the problem, but it is unlikely to be satisfactory, since the user likely does not want the rules to conflict. (And the user probably wants to retain the boundary.)

Two real solutions exist. One is to split the parent rule. The other is to change the child module so that there are no rules which must execute between the methods.

Splitting the parent rule

The parent rule can be split into two parts, one which calls `method1` and one which calls `method2`. This can be done automatically with the `(*split*)` attribute or it can be done manually by rewriting the rule.

For example, automatically splitting would look like this:

```
rule top_rule;
   (* split *)
   if (p)
      sub.method1;
   else
      sub.method2;
endrule
```

Manual splitting might look like this:

```
rule top_rule_p (p);
   sub.method1;
endrule
rule top_rule_notp (!p);
   sub.method2;
endrule
```

Removing the rule dependency in the child

The other option is to remove the execution order dependency of the rule (or rules) in the child module. This can be done in two ways: either you force the rule to conflict entirely with one of the methods (and thus it can never fire in the same cycle) or you remove the execution ordering so that they can fire in a different order. This requires analyzing the schedule of the child module and understanding why the rule has to execute after `method1` and before `method2`.

For instance, the rule could be reading and writing state which is used by both methods. Removing some method calls from the rule could prevent the execution order requirement. Another possibility

is that the rule can be split into two rules: one which shares state with `method1` and one which shares `method2`. However, this is not always possible.

The correct solution will depend on the specific design.

Problem #030: Two rules use methods which have a rule that executes between them

This KPNS entry is similar to entry #029, except that instead of the method calls being inside one rule they are in two rules.

The user may encounter any of the three following errors:

```
Error: "Test.bsv", line 30, column 9: (G0100)
  The rules ‘RL_top_rule_1’ and ‘RL_top_rule_3’ require dynamic scheduling,
  which is not supported by Bluesim. This is because the rules use methods
  which have a rule that executes between them in the static execution order
  of the separately synthesized submodule, but the rules must execute in the
  opposite order according to the current module’s schedule. See entry #30 in
  the KPNS document for more information and possible solutions.
  The methods and the rules between them are as follows:
    (sub.method2, sub.method1)
      RL_sub_rule
  The execution order path is as follows:
    ‘RL_top_rule_1’ -> ‘RL_top_rule_2’ -> ‘RL_top_rule_3’
  The relationships were introduced for the following reasons:
    (RL_top_rule_1, RL_top_rule_2)
    execution order because of calls to rg12.write vs. rg12.read
    (RL_top_rule_2, RL_top_rule_3)
    execution order because of calls to rg23.write vs. rg23.read

Error: "Test.bsv", line 13, column 9: (G0101)
  The rules ‘RL_top_rule_B’ and ‘RL_top_rule_A’ require dynamic scheduling,
  which is not supported by Bluesim. This is because the rules use methods
  which have a rule that executes between them in the static execution order
  of the separately synthesized submodule, but some pairs must execute in one
  order and some pairs must execute in the opposite order. See entry #30 in
  the KPNS document for more information and possible solutions.
  The methods and the rules between them which must execute in the forward
  direction are as follows:
    (sub1.method2, sub1.method1)
      RL_sub_rule
  The methods and the rules between them which must execute in the reverse
  direction are as follows:
    (sub2.method1, sub2.method2)
      RL_sub_rule

Error: "Test.bsv", line 2, column 8: (G0116)
  This module requires dynamic scheduling, which is not supported by Bluesim.
  This is because there are two or more pairs of rules which use methods which
  have a rule that executes between them in the static execution order of the
  separately synthesized submodule, but, when these orderings are taken into
```

```
account, a loop results in the current module's schedule. See entry #30 in
the KPNS document for more information and possible solutions.
The ordering loop is as follows:
  `RL_r4' -> `RL_r1' -> `RL_r2' -> `RL_r3' -> `RL_r4'
The relationships were introduced for the following reasons:
  (RL_r4, RL_r1) execution order because of calls to rg2.write vs. rg2.read
  (RL_r1, RL_r2)
  execution order because of calls to methods with rules between them:
    sub1.b vs. sub1.a
  (RL_r2, RL_r3) execution order because of calls to rg1.write vs. rg1.read
  (RL_r3, RL_r4)
  execution order because of calls to methods with rules between them:
    sub2.b vs. sub2.a
```

This occurs when a design is taking advantage of the fact that separately-synthesized modules have separate schedules. In an individual schedule, BSC insists that there be a static execution order of all rules and methods. By static, we mean that if rule A and rule B fire together in the same clock cycle, they will always fire in the same order.

When BSC generates separate Verilog modules, each module has its own schedule. There is no longer one static ordering of rules, but two static orders. This opens up the possibility that rule A from a parent module and rule B from a child module will execute in different orders in different clock cycles, because there is no global scheduler which is enforcing the order.

In KPNS entry #029, we explain that this can happen when a submodule has two methods between which a rule must be executed. For example, inside the submodule `sub`, there is a rule `sub_rule` which must execute after `method1` and must execute before `method2`. This normally enforces an execution order that any rule in the parent module which calls `method1` must execute before any rule which calls `method2`. However, when two rules conflict or when the two rules have mutually exclusive predicates, then the scheduler ignores any execution order between them, because it is irrelevant — the two rules can never execute in the same cycle.

For example, consider these two rules:

```
rule top_rule_1 (!p);
   sub.method2;
   // other actions
   $display(rg12);
endrule

rule top_rule_3 (p);
   sub.method1;
   // other actions
   reg23 <= True;
endrule
```

These rules have mutually exclusive predicates, so the scheduler for the module is free to ignore any execution order requirements between them, because they will never execute in the same cycle.

However, if we add a third rule into the mix, which can execute sometimes with rule `top_rule_1` and sometimes with `top_rule_3`, then an execution order is enforced by transitivity. For example:

```
rule top_rule_2;
   reg12 <= rg23;
endrule
```

This rule enforces an execution order (`top_rule_1` before `top_rule_2` before `top_rule_3`) which is in the opposite direction of the order enforced by calls to `method1` and `method2`. Dynamic scheduling occurs as a result: When `top_rule_1` executes, rule `sub_rule` (in the submodule) has to execute before it. But when rule `top_rule_3` executes, then rule `sub_rule` executes after it. Thus, the location of `sub_rule` in the execution order is not static.

Note that this feature only happens when the submodule is separately synthesized. If the synthesis boundary is removed, then an execution order cycle is detected between all four rules: `top_rule_1` before `top_rule_2` before `top_rule_3` before `sub_rule` before `top_rule_1`. In that case, the compiler will break the cycle by forcing two consecutive rules to conflict.

This feature, of separately-synthesized modules, is supported by the Verilog which BSC generates. However, since the behavior changes when the synthesis boundary is removed, it is recommended that designers not rely on it.

Furthermore, Bluesim does not support this feature. So designs which are written this way cannot be simulated with Bluesim.

When a design like this is compiled for the Verilog backend, a warning is issued and Verilog is generated, but it is recommended that the user change his design. When the design is compiled for Bluesim, BSC exits with an error.

Solutions

Removing the synthesis boundary is one way to avoid the problem, but it is unlikely to be satisfactory, since the user likely does not want any rules to conflict. (And the user probably wants to retain the boundary.)

Two real solutions exist. One is to remove the execution order dependency between the parent rules, which can be done by splitting the rules or changing the actions inside them. The other solution is to change the child module so that there are no rules which must execute between the methods.

Adjusting the parent rules

There are many way to remove the execution order in the parent module, depending on what is causing the order. It requires analyzing the schedule of the module to understand why the order is created. See the section below (on removing the execution order in the child module) for more guidance.

One way to remove the execution order dependency is to remove unnecessary shared state from some rules. In the given example, this might mean removing the `$display` statement from rule `top_rule_1`, so that it is not sharing register `rg12` with `top_rule_2`.

Another possibility is to split the rule into two pieces, one which calls the submodule method and one which performs the other actions which are the cause of the execution order dependency. For example, it may be possible to split rule `top_rule_3` into two pieces:

```
rule top_rule_3a;
   sub.method1;
endrule
rule top_rule_3b;
   rg23 <= True;
endrule
```

This option is only possible if the two actions are not required to execute atomically.

Removing the rule dependency in the child

The other option is to remove the execution order dependency of the rule (or rules) in the child module. This can be done in two ways: either you force the rule to conflict entirely with one of the methods (and thus it can never fire in the same cycle) or you remove the execution ordering so that they can fire in a different order. This requires analyzing the schedule of the child module and understanding why the rule has to execute after `method1` and before `method2`.

For instance, the rule could be reading and writing state which is used by both methods. Removing some method calls from the rule could prevent the execution order requirement. Another possibility is that the rule can be split into two rules: one which shares state with `method1` and one which shares `method2`. However, this is not always possible.

The correct solution will depend on the specific design.

Problem #031: Bluesim switches (-cc, -ss, etc.) have been removed

Bluesim used to support debugging switches for dumping clock, state and rule traces during simulation. In recent versions of Bluesim, the -cc, -s and other similar options have been removed.

Solutions

The functionality of the deprecated options is provided in a more flexible form using the new Bluesim scripting functionality (see User Guide section 8.3).

Typically, the user would execute their Bluesim model using the `-f debug.cmd` option and refine the contents of the `debug.cmd` script to focus on just the information relevant to them.

This script-based methodology is much more flexible than a small set of fixed functions controlled by command-line switches:

- The user can focus debugging output to a particular region of simulation time.
- Debugging output can be displayed differently under different conditions.
- State dumps can be limited to only the values relevant to the current debugging task.

Controlling Simulation

Simulation is controlled via the `Bluesim::sim` command. The command is part of the Bluesim tcl package, provided by Bluespec, see *User Guide* for more detail. Tcl packages are loaded with the tcl `package` command, and then (optionally) commands are imported from the package's namespace using the `namespace` command. The following examples assume that these tcl commands have been executed either in a script or during a command session.

```
package require Bluesim
namespace import ::Bluesim::*
```

Simulation can be controlled using the Bluesim `sim run`, `sim runto`, `sim step` and `sim nextedge` commands.

To run a simulation to completion and then print the elapsed simulation time, one could use the following script:

```
sim run
puts [sim time]
```

To do something on each tick of a clock, a simple `while` loop suffices:

```
while {true} {
  puts "CLOCK EDGE: [sim clock]"
  if [catch {sim step}] {break}
}
```

Debugging actions can be targeted at a specific simulation period:

```
# dump VCD waveforms for 1000 cycles starting at time 4207
sim runto 4207
sim vcd on
sim step 1000
sim vcd off
```

It is also simple to run the simulation until a particular time or condition with a debugging action at each step:

```
while {[sim time] < 1000} {
  puts "CLOCK EDGE: [sim clock]"
  if [catch {sim step}] {break}
}
```

Displaying State Values

Bluesim interactive commands allow simulation values to be looked up dynamically. The first step is to obtain a handle for the value using the `sim lookup` command. Then the handle can be used (repeatedly) to retrieve the simulation value using the `sim get` command.

To trace the state and method values in the canonical GCD example, one could use this script:

```
sim cd gcd

# get handles for signals we're interested in
foreach name [list EN_start start_num1 start_num2 RDY_result result the_x the_y] {
  set hdl_of($name) [sim lookup $name]
}

# step and watch values
while {true} {
  if [catch {sim step}] {break}
  puts "---- State at [sim time] ----"
  foreach name [array names hdl_of] {
    set label "$name:"
    set value [sim get $hdl_of($name)]
    puts [format "%-15s %s" $label $value]
  }
  puts "--------------------"
}
```

Scripts can also be used to collect debugging information targeted at specific simulation conditions. Again using the canonical GCD example, one could trace the sequence of start method calls using this script:

```
sim cd gcd

# get handles for start method enable and arguments
set hdl_of(EN_start)   [sim lookup EN_start]
set hdl_of(num1)       [sim lookup start_num1]
set hdl_of(num2)       [sim lookup start_num2]

# log start method calls
while {true} {
  if [catch {sim step}] {break}
  if {[sim get $hdl_of(EN_start)] == "1'h1"} {
     set num1 [sim get $hdl_of(num1)]
     set num2 [sim get $hdl_of(num2)]
     puts "[sim time]: start($num1,$num2)"
  }
}
```

Displaying Clock Boundaries

Displaying clock boundaries interspersed with display output from a model is very simple when there is only a single clock domain:

```
while {true} {
  if [catch {sim step}] {break}
  puts "/CLK at [sim time]"
}
```

In an MCD design, the `sim step` command steps in cycles of the current clock domain. To advance to the next edge without regard to its domain, use the `sim nextedge` command:

```
while {true} {
  if [catch {sim nextedge}] {break}
  puts "---- [sim time] ----"
}
```

The data returned by the `sim clock` command can be parsed and used to print a more readable display of the clock activity, similar to the output of the old `-cc` option:

```
# show the clock edges at the current simulation time
proc showclk {} {
  set t [sim time]
  set clks [sim clock]
  foreach clk $clks {
    set name [lindex $clk 2]
    set cycles [lindex $clk 7]
    set val [lindex $clk 8]
    set edge_at [lindex $clk 9]
    if {$edge_at == $t} {
       if {$val == 1} then {
```

```
          puts "/$name ($cycles) at $edge_at"
       } else {
          puts "\\$name ($cycles) at $edge_at"
       }
    }
  }
}

while {true} {
  if [catch {sim nextedge}] {break}
  showclk
}
```

Problem #032: Unexpected use of uninitialized value error (G0028)

The compiler unexpectedly complains about a use of an uninitialized value. There are two common reasons this complaint might be unexpected:

1. The code in question worked in a previous compiler release. (The compiler's handling of uninitialized values changed between the 2008.06.E and 2008.11 releases.)
2. The value seems to be initialized along all possible execution paths.

Solutions

The reason for the compiler change was to improve the detection of uses of uninitialized values in compound types (e.g. vectors, structures, `Bit#(n)`, etc.). Previous compiler releases would (silently) initialize a compound type to an undefined value whenever any of its parts were assigned, incorrectly hiding the existence of uninitialized parts of the value. The compiler now tracks initialization component-by-component, so it can detect problems that were hidden in past.

Consider the following code:

```
Bit#(2) x;
x[0] = 0;

rule test;
  $display(x)
endrule
```

In older Bluespec releases, the compiler merely saw that the variable `x` was assigned to before it was used and filled out the unassigned components of `x` with undefined values. Now the compiler correctly sees that while `x[0]` has been assigned, `x[1]` is initialized and (correctly) complains about that use. The solution, in this case, is to initialize the components of a variable before using them. Similar considerations apply to Vectors, structures and other compound types.

The second reason the compiler might complain about a use of an uninitialized value is that it does not attempt to prove that all dynamic execution paths correctly initialize a variable.

Consider the following code:

```
Reg#(Int#(32)) r <- mkRegU;

Maybe#(Int#(32)) x;

if(r > 0)  x = tagged Valid r;
if(r <= 0) x = Invalid;
```

The compiler doesn't attempt to prove that the two predicates `r > 0` and `r <= 0` cover all possible values of `r` (and, hence, guarantee that `x` is always safely initialized). That means it will (incorrectly) complain about an uninitialized value when `x` is used. The recommended workaround in this case is to initialize the variable to a sensible default (`Invalid` might work in the example above) before changing that default with conditional assignments. Please note that this limitation also existed in releases before 2008.11, but was often masked by the previous issue (not tracking the initialization of parts of compound data).

Below is another example in which the compiler complains of an uninitialized value. Like the previous example, the compiler does not attempt to prove that `x` is set in all branches of the case statement. Again, the compiler will (incorrectly) complain about an uninitialized value when `x` is used. The recommended workaround in this case is to initialize the variable `x` when it is declared, or to add a default clause in the case expression.

```
Reg#(Maybe#(Int#(32))) mx <- mkRegU;

Int#(32) x ;
case (mx) matches
    tagged Valid .v: x = v;
    tagged Invalid:  x = 0;
endcase
```

Index