
1

Multiple Analog Constructs

2 **Abstract**

3

4

5

6

7

Support in the Verilog-AMS language for multiple analog constructs (or analog blocks) within one module has impact on numerous aspects of the language. This document aims to provide a proposal for handling multiple analog constructs, and to define clearly the selection criteria by which the current solution has been chosen over other possible solutions.

8 **Version**

4

9 **Date**

April 17, 2007

10 Table of Contents

11	1	Introduction	3
12	1.1	Limitations	3
13	1.2	Guidelines	3
14	2	Rationale.....	5
15	2.1	Why Multiple Analog Constructs?.....	5
16	2.2	Generate Constructs	5
17	2.3	<u>Latency</u>	Error! Bookmark not defined.
18	2.4	<u>Sharing Variables</u>	Error! Bookmark not defined.
19	3	Multiple Analog Constructs	8
20	3.1	Concurrent Analog Constructs.....	8
21	3.1.1	Race Conditions	9
22	3.1.2	Switch Branches	11
23	3.1.3	Out-of-Module References (OOMRs)	14
24	3.2	Continued Analog Constructs.....	15
25	3.2.1	Generate constructs and continuation.....	15
26	3.2.2	Analog construct continuation	16
27	4	Examples	19
28	4.1	Concurrent Analog Constructs.....	19
29	4.1.1	Upward hierarchical referencing	19
30	4.1.2	Race conditions.....	19
31	4.2	Continued Analog Constructs.....	20
32	4.2.1	Continued analog construct with preceding qualifier	20
33	4.2.2	Continued analog sequential block.....	22
34	4.2.3	Continued analog named sequential block	22
35	4.2.4	Concatenate construct	23
36	4.2.5	Concatenate compiler directive.....	25

37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58

1 Introduction

The Verilog-AMS language definition as to be defined in the Language Reference Manual version 2.3 is intended to align the Verilog-AMS language with the IEEE 1364-2005 standard for the Verilog hardware description language.

One of the items within the Verilog language which is proposed to be adopted in the Verilog-AMS 2.3 LRM as well is the concept of generate constructs. These constructs allow a model developer to create models whose structure depends on instance parameter values providing a powerful structural modelling approach. To be able to combine the behavioural and structural modelling and hence get the same advantage of generate constructs in the Verilog-AMS language as is available in the digital Verilog, generate constructs should be able to contain analog constructs (blocks). The consequence of allowing analog constructs within generate blocks is that multiple analog blocks within the same module needs to be supported.

SystemVerilog as described in the IEEE 1800-2005 standard takes the generate construct directly from the IEEE 1364-2005 standard.

This document tries to collect the various consequences of support for multiple analog blocks and how to resolve possible conflicts and ambiguities.

59

1.1 Limitations

60
61

The options for support of multiple analog blocks is restricted by the following set of boundary conditions:

62
63
64
65
66
67
68
69
70
71
72
73
74
75

1. Existing Verilog-AMS modules defined according to LRM 2.2 should have the same behaviour – the option to support multiple analog blocks should have no impact on the support of single analog blocks per module, either according to the new specification or from legacy code.
2. No new restrictions should be imposed on language items that have no direct relation with the support for multiple analog blocks. It is possible that in case of multiple analog blocks use of certain language items needs to be restricted when they are used in multiple analog blocks.
3. No new constructs, syntax, or semantics should be introduced that are incompatible with either digital Verilog (1364-2005) or SystemVerilog (1800-2005). No new areas of ambiguity between these three languages should be created.

76

1.2 Guidelines

77
78

Apart from the above limitations there are also a couple of guidelines that should be taken into consideration:

79
80
81
82
83
84

1. No new keywords should be introduced unless deemed absolutely necessary.
2. If new keywords are introduced, let them be in line with the existing syntax and keyword definitions of Verilog-AMS.
3. If new keywords are introduced, they should not be incompatible with keywords in with either digital Verilog (1364-

85 2005) or SystemVerilog (1800-2005).

86 4. Multiple analog blocks should also be supported in Verilog-A,
87 the analog-only subset as defined in Annex E of the Verilog-
88 AMS LRM.

89 5. The approach should be as much as possible implementation
90 independent. The specification should be complete and clear
91 enough so that there is little chance for ambiguity.

92

93
94
95

2 Rationale

In this section some of the reasons for support of multiple analog blocks in the Verilog-AMS language are presented.

96

2.1 Why Multiple Analog Constructs?

97
98
99
100
101
102
103
104

In the Verilog-AMS LRM version 2.2, section 6.1 it is explicitly mentioned that per module only one analog block can be allowed. Also section 7 on hierarchical structures mentions this fact. For the upcoming revision 2.3 of the standard the aim is to match the AMS extensions of the Verilog language up with a more recent version of the IEEE 1364 standard: the 2005 revision of that language to be specific. One of the more influential extensions of that version of the language is the support for generate constructs.

105
106
107
108
109
110
111

Generate constructs allow a module writer to conditionally add one or more structural or behavioral descriptions. This is not entirely new to the AMS language. In Verilog-A 1.0 the generate statement – which has been obsoleted since version 2.0 – provided similar functionality for structural descriptions. The 1364-2005 generate constructs however, are more versatile and exist for both analog and digital blocks, and hence mixed-signal blocks.

112

2.2 Generate Constructs

113
114
115
116

To support generate constructs for analog behavioral descriptions it will be necessary to be able to handle multiple analog blocks within the same module. To illustrate this, here's an example of an analog behavioral model using a 1364-2005 loop generate construct.

117

Example 1. Loop generate construct example.

118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148

```
module adc (in, out);
  parameter integer bits = 8 from [0:inf];
  parameter real fullscale = 1.0;
  parameter real dly = 0.0;
  parameter real ttime = 10n;
  input in;
  output [0:bits-1] out;
  electrical in;
  electrical [0:bits-1] out;
  localparam thresh = fullscale/2.0;
  real sample, value;
  genvar i;

  analog
    sample = V(in);

  generate
    for (i = bits - 1; i >= 0; i = i - 1)
      analog begin
        if (sample > thresh) begin
          value = fullscale;
          sample = 2.0 * (sample - thresh);
        end
        else begin
          value = 0.0;
          sample = 2.0 * sample;
        end
        V(out[i]) <+
          transition(value, dly, ttime);
      end
    endgenerate
endmodule
```

149
150 `endmodule // adc`
151 In this example of an analog-to-digital converter the number of bits
152 of the digital output can be set using a parameter. The generate
153 construct will then calculate the correct conversion of the analog
154 input. The model does not have a clock so the conversion may
155 change on every change of the input signal `V(in)`.

156 There are a couple of issues involved in the above example, so it is
157 only meant as an example of the generate construct syntax, not of
158 its semantics. These will be covered later.

159 The other form of the 1364-2005 generate construct is the
160 conditional generate construct that has the form of an `if`
161 statement. This allows a user to make part of a behavioral or
162 structural model optional. To illustrate this, here's an example of
163 an analog structural model using a 1364-2005 conditional generate
164 construct.

165 **Example 2.** Conditional generate construct example.

```
166 module pipeline_adc (in, out)
167     parameter integer bits = 8 from [0:inf);
168     parameter real fullscale = 1.0;
169     parameter real dly = 0.0;
170     parameter real ttime = 10n;
171     inout in;
172     inout [0:bits-1] out;
173     electrical in;
174     electrical [0:bits-1] out;
175     real value;
176
177     analog begin
178         if (V(in) > fullscale/2.0)
179             value = fullscale;
180         else
181             value = 0.0;
182         V(out[bits-1]) <+ transition(value, dly, ttime);
183     end
184
185     generate
186         if (bits > 1)
187             begin
188                 electrical n1;
189
190                 analog
191                     V(n1) <+
192                         2.0 * fullscale * (V(in) - value);
193
194                 pipeline_adc
195                     #(.bits(bits-1), .fullscale(fullscale))
196                     section (n1, out[0:bits-2]);
197             end
198     endgenerate
199
200 endmodule // pipeline_adc
```

201 In the above example the model is recursive: it actually
202 instantiates itself. Just as with the first example for the loop
203 generate construct some other semantic issues may appear once
204 the model is examined in detail. Again, this will be covered later.

205 2.3 Latency

206 Compact device models are among the largest and most complex
207 models to be written in the Verilog-A language. To use such models

208 in a simulation and maintain an acceptable performance some
209 compiler optimizations are required. An important way to achieve
210 an acceptable performance is to activate various parts of the model
211 based on the level of activity. A part of the model that considers
212 parameter verification often needs to run only once; another part
213 related to calculating the terminal currents must probably be run
214 every time step. By logically grouping the statements of the model
215 in separate sections the compiler can quickly decide which parts of
216 the model have to be evaluated at any given step in the simulation.

217 Some compilers* have some limited ability to take this latency into
218 account by using – by convention – specific named sequential
219 blocks within a single analog construct. Some other compilers are
220 able to derive the need for named block evaluation by constructing
221 a dependency tree. One could consider that a more natural and
222 portable way would be to use multiple analog blocks for this.

223 However, a more fine-grained approach would not need the user to
224 partition the code in separate analog blocks to make optimal use of
225 latency in the model description. By determining the dependency
226 on a statement-by-statement basis which statements should be
227 evaluated, the simulator can achieve optimal performance. To be
228 able to deal better with dynamic situations one could instead use
229 sensitivity lists with respect to connected signals and internal state
230 which is quite similar to what is done in digital Verilog. Either way,
231 this precludes the need for multiple analog constructs by which the
232 user is just providing hints on what might be optimal (where he
233 could be wrong!). In that sense, improved handling of latency
234 should not be considered as a driver for the support of multiple
235 analog constructs.

236 **2.4 Sharing Variables**

237 From a semantic point-of-view multiple analog constructs appear
238 very similar to multiple analog module instances inside another
239 (analog) module. Such instances are essentially considered to be
240 evaluated concurrently – they can even access their respective
241 continuous states through out-of-module references (OOMR).
242 However, what they can not do is access each other variables.
243 Verilog-AMS 2.2 but also IEEE 1364-2005 and IEEE 1800-2005 do
244 not allow access to variables values outside the module.

245 Multiple analog constructs share the module level scope: they have
246 access to the same terminals, parameters, branches, and therefore
247 also to the variables declared in the module heading. For models
248 that require a certain level of independence of the analog
249 continuous processes but benefit from shared interaction through
250 access to their variables the use of multiple analog constructs
251 would offer advantages over separate analog instances that can
252 only communicate through nodes and terminals or a single analog
253 construct that is unable to provide the concurrency.

254 Consequentially, if this feature is not used the multiple analog
255 constructs behave as independent analog instances.

* The Open Source ADMS Verilog-A compiler has this ability for compact device models.

3 Multiple Analog Constructs

In this section the various options with respect to multiple analog blocks are presented. Impact of various analog, digital and mixed-signal constructs are addressed for these options.

3.1 Concurrent Analog Constructs

The most straightforward approach to analog constructs is to consider them as an analog equivalent of the always and initial constructs from digital Verilog as defined by IEEE 1364-2005. These have the following characteristics according to section 9.9 of the standard:

- The initial and always constructs are enabled at the beginning of a simulation.
- The initial construct shall execute only once, and its activity shall cease when the statement has finished.
- In contrast, the always construct shall execute repeatedly. Its activity shall cease only when the simulation is terminated.
- There shall be no implied order of execution between initial and always constructs. The initial constructs need not be scheduled and executed before the always constructs.
- There shall be no limit to the number of initial and always constructs that can be defined in a module.

For concurrent analog constructs there is no explicit distinction between constructs that (need to) execute only once and constructs that execute at every step in the analog simulation cycle. Therefore translating the above description of initial and always constructs the characteristics of concurrent analog constructs would be:

- Concurrent analog constructs are enabled at the beginning of a simulation in all stages of the analog simulation loop.
- Concurrent analog constructs shall execute as often as required by the statements in the construct, but restricted by the requirements of the analog simulation cycle.
- There shall be no implied order of execution between the concurrent analog constructs in a module.
- There shall be no limit to the number of concurrent analog constructs that can be defined in a module.

Optionally we can add a rule to make the results of concurrent evaluation of analog constructs more predictable:

- The statement in a concurrent analog construct is evaluated atomically – if two concurrent analog constructs both contain a sequential block with multiple statements then first all statements in one sequential block are evaluated and only then all statements in the other sequential block are evaluated.

The above optional rule becomes especially helpful when values are assigned to variables from multiple concurrent analog constructs.

Considering the above characteristics to define a concurrent analog

303 construct, it is required that we address the following items in
304 combination with multiple concurrent analog constructs:

- 305 • unnamed branches
- 306 • race conditions
- 307 • switch branches
- 308 • out-of-module references (OOMRs)

309 **3.1.1 Unnamed Branches**

310 In a Verilog-AMS module with a single analog block it is not
311 necessary to consider whether an unnamed branch is part of the
312 module or part of the analog block. As unnamed branches can only
313 appear in analog blocks the distinction is irrelevant.

314 This is different for multiple analog blocks. If an unnamed branch is
315 defined at the module level, then contributions in multiple analog
316 constructs should add up when solving the DAE system. However,
317 if an unnamed branch is defined at the concurrent analog construct
318 level an error may be produced when constant potentials are forced
319 – equivalent to two parallel independent voltage sources. The
320 handling of parallel independent voltage sources is
321 implementations specific: some solvers allow them and will provide
322 a solution, while others do not accept them and generate a run-
323 time error.

324 It should be noted that the difference only affects potential
325 contributions: flow contributions to unnamed branches always add
326 up in the same manner. Essentially the choices for defining their
327 behavior are:

- 328 • Unnamed branches are defined at the module level. In this case
329 the potential contribution to an unnamed branch for a node pair
330 in one concurrent analog construct adds up to the contribution
331 to an unnamed branch for the same node pair in another
332 concurrent analog construct.
- 333 • Unnamed branches are defined at the analog construct level. In
334 that case the potential contribution to an unnamed branch for a
335 node pair in one concurrent analog construct are equivalent to
336 parallel potential branches whose handling is implementation
337 specific.

338 To see why it is undesirable to associate unnamed branches with
339 the block rather than the module consider the model of a nonlinear
340 lossy inductor. Assume that there is a logical separation of the
341 model into code that describes the nonlinear inductor and code that
342 describes the loss. If these are put into one analog module (and
343 assuming that they both contribute to the potential of the branch)
344 then they would go in series. However, if they were each in their
345 own analog blocks they end up in parallel, which is not the
346 intention.

347 Presumably this issue could be addressed in one of two ways.
348 Either they could be forced to be combined in series by creating a
349 named branch and have them both contribute to that. Or both
350 effects could be combined into a single analog block and separated
351 by placing them both in their own named blocks. The former is
352 undesirable because it forces the use of named branches. The
353 latter is undesirable because it constrains the modularity (all

354 analog behavior must be adjacent, and cannot be intermingled with
355 the event-driven behavior).

356 Unnamed branches should not be associated with analog blocks:
357 this will result in branches from different blocks being combined in
358 parallel.

359 In a single module definition separate analog constructs should act
360 identical to separate module instances with the only added feature
361 the ability to share module-level variables.

362 The following is recommended:

- 363 1. associating the unnamed branches with the module;
- 364 2. allow named branches to be declared within (named) analog
365 blocks.

366 An additional option is to allow contributions to unnamed branches
367 to appear in only one analog construct. Consequentially, the other
368 analog constructs will have to use named branches if they want to
369 contribute to a particular potential or flow. However, this makes
370 use of multiple analog constructs in conjunction with generate
371 constructs more complicated.

372 **3.1.2 Race Conditions**

373 Race conditions are situations where a value is assigned to the
374 same variable in multiple concurrent processes. In the case of
375 concurrent analog constructs these race conditions may occur when
376 assignment to a particular module-level variable is done in multiple
377 concurrent analog constructs. Through upward hierarchical
378 references race conditions can also occur when assigning to a
379 variable in a lower scope, again in multiple concurrent analog
380 constructs.

381 There are a couple of ways to deal with race conditions:

- 382 1. Disallow the occurrence of race conditions by requiring that
383 assignment to a variable can occur in one and only one analog
384 construct.
- 385 2. Allow only explicit occurrence of race conditions by requiring
386 that any assignments that would introduce a race condition are
387 marked with a relevant attribute to allow a race condition, for
388 example (`* race_condition *`).
- 389 3. Allow race conditions to occur but require the compiler to issue
390 a warning. As an extension the warning can be switched off by
391 marking the violating assignment with a relevant attribute such
392 as the example above.
- 393 4. Allow race conditions to occur but have the runtime
394 environment produce an error when an assignment is made to
395 the same variable in multiple concurrent analog constructs at
396 the same step in the analog simulation cycle.
- 397 5. Make a distinction based on the execution of the statement:
 - 398 a. If a variable is assigned a value continuously in an
399 analog construct, it is owned by that construct and its
400 value cannot be assigned in another block (though it can
401 be read).
 - 402 b. A variable assigned a value within an event statement

403 (@) that is contained in an analog construct is not said
404 to be assigned a value continuously.

405 c. A variable that is not assigned continuously, can be
406 assigned a value by any analog, always, or initial
407 construct.

408 Although option (4) gives most freedom it may lead to situations
409 that are hard to control, unless the module developer has a good
410 sense about when what concurrent analog construct will be
411 activated.

412 Option (1) guarantees that a model will generate the same results
413 for all implementations. However, as the digital Verilog seems to
414 get by quite well with fewer restrictions this seems a bit too
415 restrictive for mixed-signal use.

416 Options (2) and (3) are nearly equivalent, but they essentially give
417 the same level of freedom as option (4).

418 Option (5) tries to prevent the occurrence of a continuous race
419 condition but gives one the freedom to share variables more fully.

420 3.1.2.1 Copy-in copy-out behavior

421 It has been argued that in the analog simulation cycle no true race
422 conditions occur, as the simulation of the contribution statement
423 occurs concurrently in the matrix solver. However, the following
424 example will make the value that is printed very much dependent
425 on the order in which the separate concurrent analog blocks are
426 being executed.

427 Example 3. Race condition example

```
428 module race (out);  
429 inout [0:1] out;  
430 electrical [0:1] out;  
431 real x;  
432  
433     analog  
434     begin  
435         x = 1.0;  
436         V(out[0]) <+ x;  
437         x = 2.0;  
438         V(out[1]) <+ x;  
439     end  
440  
441     analog  
442     $strobe(.x = %g., x);  
443  
444 endmodule // race
```

445 The question is: what the value is that is printed by the \$strobe
446 task in the second analog construct? Even if the optional rule
447 presented in section 3.1 is taken into account the arbitrary order in
448 which the concurrent analog constructs may be evaluated might
449 either return 0.0 or 2.0.

450 For the analog simulation cycle in which the bodies of the various
451 analog blocks may be evaluated multiple times before convergence
452 is achieved this may lead to non-convergence. To assure the ability
453 to achieve convergence even in the presence of race conditions,
454 the best option is copy-in, copy-out behavior for the various
455 concurrent analog blocks. This means that the values of all module-
456 level variables are copied into each of the analog blocks at the start
457 of the evaluation of their bodies, and copied out at the end of the

458 evaluation. The consequence is that all analog blocks using the
459 same variable values in the evaluation, preventing the occurrence
460 of race conditions.

461 Related to [Example 3](#) the output from the \$strobe will always be
462 the value x had at the end of the previous accepted time step. In
463 the case of the initial step this will always be 0, independent of the
464 fact that the analog block above it could have been evaluated first.

465 On the downside: this disallows the reuse of a module-level
466 variable value calculated in another concurrent analog block for the
467 same time step. To solve this the relation to the analog block in
468 which the calculation is done should be made explicitly by making
469 it a local variable and using a hierarchical reference to access that
470 variable.

471 **Example 4.** Race condition example

```
472 module rerace (out);  
473 inout [0:1] out;  
474 electrical [0:1] out;  
475  
476     analog  
477         begin : block1  
478             real x;  
479             x = 1.0;  
480             V(out[0]) <+ x;  
481             x = 2.0;  
482             V(out[1]) <+ x;  
483         end  
484  
485     analog  
486         $strobe(•x = %g•, block1.x);  
487  
488 endmodule // race
```

489 As a consequence, this should define an order of evaluation of the
490 related blocks. If the order of evaluation cannot be uniquely
491 determined an error should be produced.

492 A bigger problem with copy-in, copy-out behavior is that it is no
493 longer possible to collect the initialization code in a separate block,
494 as that code would not become available to the other (dependent)
495 blocks until the next step. However, as such initialization is merely
496 an exploitation of latency which should probably not be addressed
497 by multiple analog constructs in the first place (see section 2.3),
498 the problems with copy-in, copy-out are not directly show-
499 stoppers.

500 **3.1.3 Switch Branches**

501 First, we need to establish what exactly is a switch branch.

502 According to section 5.1.5 source branches have the ability to
503 switch between being potential and flow sources. To switch a
504 branch to being a potential source, assign to its potential. To switch
505 a branch to being a flow source, assign to its flow.

506 A discontinuity of order zero (0) is assumed to occur when the
507 branch switches.

508 According to section 5.1.6 if no value is assigned to a branch, the
509 branch flow is set to zero (0).

510 Finally, section 5.3.1.3 says that contributing a flow to a branch
511 which already has a value retained for the potential results in the

512 potential being discarded and the branch being converted to a flow
513 source. Conversely, contributing a potential to a branch which
514 already has a value retained for the flow results in the flow being
515 discarded and the branch being converted into a potential source.
516 It is illegal to contribute to an external switch branch from within
517 an analog block.

518 The issue of switch branches with concurrent analog constructs is
519 threefold:

- 520 1. If a branch is contributed to in multiple analog constructs within
521 the same module, and a branch is given a flow value in one
522 concurrent analog construct and a potential value in another,
523 should that branch be considered a switch branch?
- 524 2. If there is no implied order in the execution of concurrent
525 analog constructs and a branch is given a flow value in one
526 construct and a potential value in another, what will be the
527 result?
- 528 3. If a branch is given either a flow value or a potential value
529 within a single analog construct, should any contribution to that
530 branch from another concurrent analog construct within the
531 same module be considered illegal as though it were a
532 contribution to an external switch branch?

533 With the above questions in mind, here are a couple of proposals
534 on how to handle concurrent switch branches:

- 535 1. Disallow concurrent switch branches: require a switch branch to
536 be given a value in one and only one concurrent analog block,
537 and disallow any contributions from outside the module (as
538 currently in the Verilog-AMS 2.2 standard) as well as from other
539 concurrent analog blocks within the same module.
- 540 2. Allow concurrent switch branches but require them to be proper
541 switch branches in each of the concurrent analog blocks in the
542 module, and require them to use the same conditional
543 expression for switching so a concurrent switch branch is well
544 defined at any time during the evaluation of the concurrent
545 analog blocks.
- 546 3. Allow concurrent switch branches but have the compiler issue a
547 warning about possible race conditions in case of contributing
548 both a flow and a potential to the same branch from different
549 concurrent analog constructs at the same step in the analog
550 simulation cycle. With this approach it would also be possible to
551 allow contributions to switch branches from external modules.

552 In all cases the definitions in the Verilog-AMS 2.2 standard for
553 branch switching according to section 5.1.5 and 5.1.6 as well as for
554 value retention according to section 5.3.1.3 need to be retained
555 and considered part of the basic language structure. Support for a
556 single unnamed branch supporting both potential and flow
557 contributions would break existing switch branch applications.

558 Option (3) seems to provide most freedom for the module
559 developer while limiting the issue of ambiguity and non-
560 deterministic behavior implied by concurrent activity. However, it
561 puts the burden in the hands of the implementor who has to
562 provide the detection and warning mechanism.

563 Option (1) seems reasonable in allowing a simple and efficient
564 definition of a switch model – the reason the switch branch exists

565
566
567
568

in the first place – while closing the door on complicated behavior that is more likely to occur by accident than by design. As a third argument it can be stated that option (1) is the easiest to implement.

569
570
571
572
573
574
575
576

3.1.3.1 Switch branches and back-annotation

One of the uses of having contributions from multiple analog processes, even those outside the module scopes, is in back annotation of parasitic devices into a Verilog-AMS model hierarchy. As it is not known to the model maker whether a branch that is contributed to by a parasitic device is a potential or a flow branch that branch should not become a switch branch but rather a parallel branch.

577

Example 5. Switch/parallel branch example

578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593

```
module cap_switch (n1, n2);  
  inout n1, n2;  
  electrical n1, n2;  
  parameter real cap = 12f from (0:inf);  
  parameter integer isOn = 0 from [0:1];  
  
  analog  
    if (isOn)  
      V(n1, n2) <+ 0;  
    else  
      I(n1, n2) <+ 0;  
  
  analog  
    V(n1, n2) <+ idt(I(n1, n2), 0)/cap;  
  
endmodule // cap_switch
```

594
595

In [Example 5](#) there will be two parallel unnamed branches, one for each analog construct.

596
597
598
599
600
601

A slightly more involved alternative is to have a parallel unnamed branch only when there is a mismatch in the branch natures within the same discipline. This would allow two potential branches still to add up correctly. However, if one of the branches is a true switch branch always a parallel branch should be created as switching the branch could create a mismatch.

602
603

Possibly the advantages of this approach cannot offset the disadvantages in more complicated bookkeeping

604

3.1.4 Out-of-Module References (OOMRs)

605
606
607
608

As already shortly described in the previous subsection out-of-module references or OOMRs for short are a powerful way to add additional behaviour to an existing module, even one whose contents are not modifiable by the user.

609
610
611
612
613
614

In Verilog-AMS 2.2 it is allowed to contribute to branches anywhere in the hierarchy as long as the contribution is the same as that of the module that owns the branch. As a branch that is unassigned in a module is automatically a flow branch per section 5.1.6 of the Verilog-2.2 LRM, through an OOMR one can only contribute a flow to an unspecified branch.

615
616
617

The current language standard states that OOMRs cannot create a branch that does not exist in the module. That implies that the kind of branches that are created implicitly as a result of an analog

618 module definition is implementation dependent. However, this is
619 slightly in contest with the creation of parallel unnamed branches
620 for each contribution from a unique analog process. Probably OOMR
621 contributions should also create parallel branches.

622 The Verilog-2.2 LRM section 7.5 says that an OOMR cannot access
623 an external variable; it can only contribute to an external branch.
624 The 1364-2005 standard section 12.5 implies that any object
625 whose hierarchical name is known can be sampled or changed, and
626 that would include variables.

627 With the 1364-2005 section 12.5 extension OOMRs can generate
628 race conditions. These can be handled in essentially the same way
629 as in-module race conditions due to multiple concurrent analog
630 constructs, see section 3.1.1 of this document.

631 **3.2 Continued Analog Constructs**

632 In combination with generate constructs there are a couple of
633 situations where concurrent behavior of the multiple analog
634 constructs within a single module is not appropriate – there is a
635 certain order of execution of the analog constructs needed to
636 achieve the desired behaviour. For this a special extension to the
637 concurrent analog construct is needed: a continued analog
638 construct.

639 Continued analog constructs are executed as though all code in the
640 analog constructs' statements had been concatenated into a single
641 analog construct in the order as defined in the module.

642 Consider the `adc` example in section 2.2: the analog blocks used
643 inside the generate construct actually need to be concatenated as
644 the loop generate construct makes the analog construct reuse the
645 calculated value of the variable `sample` in the next analog
646 construct. Without the explicit ordering of a continued analog
647 construct the order of the output bits would become arbitrary: the
648 model would no longer act as a proportional analog-to-digital
649 converter.

650 **3.2.1 Generate constructs and continuation**

651 It should be realized that the only context in which continued
652 analog constructs make sense is that of the generate constructs. In
653 all other cases the analog constructs that are supposed to be
654 concatenated can be concatenated explicitly by the model
655 developer.

656 As such it should be sufficient to allow continuation only for the
657 analog blocks immediately preceding and succeeding a generate
658 construct or a series of generate constructs. Any analog constructs
659 that do not take part in the concatenation are by default concurrent
660 analog constructs and as in that case their order of definition does
661 not imply their order of evaluation they can be safely moved to
662 elsewhere in the module body. This allows multiple concurrent
663 analog constructs themselves consisting of multiple continued
664 analog constructs to coexist without requiring explicit scope
665 creation – there is no ambiguity as to which concurrent analog
666 construct a continued analog construct should be concatenated.

667 As a generate construct creates a new scope according to the
668 1364-2005 standard section 12.4, concatenations of generate

669 constructs that contain continued analog constructs should also
670 create a new scope in the concatenated analog construct such that
671 the hierarchical reference to the analog statements does not
672 change, i.e. it has to use the correct implicit naming according to
673 1364-2005 section 12.4.3. This way the extension of the generate
674 constructs with analog constructs does not change its semantics: a
675 generate construct creates a new scope, even if the generate body
676 is an unnamed sequential block.

677 **3.2.1.1 Structural and behavioral generate constructs**

678 It is possible for generate constructs to contain both analog
679 constructs and (analog) structural elements in the generate body.
680 There should be no ambiguity in referencing either of these
681 elements.

682 Accessing the structural elements of the generate body can be
683 done using the hierarchical referencing or upward hierarchical
684 referencing as defined in 1364-2005 sections 12.5 and 12.6 and in
685 1800-2005 section 19.3. As all of these together seem not to be
686 complete nor completely accurate we will for the moment stick with
687 the 1364-2005 definition.

688 For access to the analog constructs of the generate body there is
689 no ambiguity – the accessible elements in this case can only be
690 variables. To define new variables within the scope of a generate
691 block the analog construct needs to contain a named sequential
692 block to contain these definitions.

693 The extension of the generate constructs to the analog
694 environment also allows new analog nets and branches to be
695 defined within the generate block. These can be accessed using the
696 hierarchical name for the generate block.

697 **3.2.2 Analog construct continuation**

698 To tell the compiler to concatenate the analog constructs there are
699 a couple of possibilities:

- 700 1. Add a qualifier keyword before the analog construct, similar to
701 the `priority` and `unique` qualifiers for conditional statements
702 in the SystemVerilog standard (IEEE 1800-2005).
 - 703 a. This would concatenate the contents of the analog
704 construct following the qualifier keyword to the last
705 analog construct defined before the qualifier keyword. If
706 no analog construct precedes the qualifier construct it
707 concatenates to an implicitly defined empty concurrent
708 analog construct. It may also emit a warning that a
709 preceding analog construct is missing.
 - 710 b. Alternatively, if an analog construct is meant to be
711 continued it should always display the qualifier keyword
712 before the `analog` keyword. This means that a continued
713 analog construct succeeding an unqualified analog
714 construct will not be concatenated: they will be two
715 concurrent analog constructs.

716 Option (b) makes it harder to define two concurrent sets of
717 continued analog constructs, as only a new concurrent analog
718 construct will end the concatenation of one set of analog
719 constructs and start the concatenation of the other set. Option

720 (b) essentially requires that an empty concurrent analog
721 construct be inserted in between the two sets of continued
722 analog constructs.

723 The concatenation of continued analog constructs should not
724 change the order in which the statements appear in the
725 continued analog constructs nor the order in which the
726 continued analog constructs appear. The concatenation of
727 continued analog constructs should not change the hierarchical
728 referencing to items that do create a new scope inside the
729 analog constructs: the concatenation shall have no impact on
730 hierarchical referencing.

731 A special consideration should be given to named sequential
732 blocks as these create a new scope. To reference a variable at a
733 different hierarchical level within the same module it shall be
734 possible to use upward name referencing as defined in 1364-
735 2005 section 12.6. In case of multiple analog blocks it shall also
736 be allowed to access variables in a hierarchical level defined in
737 or below another concurrent analog block.

738 Possible qualifier keywords are `continue`, `concat` or
739 `concatenate`. The name `continue` is already a keyword in
740 SystemVerilog (IEEE 1800-2005, section 10.6) but its use there
741 is completely orthogonal to the use in the context of continued
742 analog constructs so a future merger of Verilog-AMS and
743 SystemVerilog should not give any problems. Nevertheless,
744 preference should be given to new keywords that do not clash
745 with existing ones.

746 2. Require that the statement inside the analog construct to
747 continue is a sequential block. This would be similar to the
748 `priority` and `unique` qualifiers for conditional statements in
749 the SystemVerilog standard (IEEE 1800-2005). A qualifier
750 keyword is added before the `begin` keyword of the sequential
751 block. This would concatenate the sequential blocks that are
752 marked this way inside multiple analog constructs.

753 To correctly handle the new scopes created by named
754 sequential blocks there are some options:

755 a. The name of a named sequential block has to be unique.
756 When concatenating the sequential blocks the named
757 sequential block is created within an overall unnamed
758 sequential block. This way the hierarchical reference to
759 the named sequential block or its contents are not
760 affected.

761 b. The name of a named sequential block does not have to
762 be unique, when concatenating sequential blocks two
763 named sequential blocks that share the same name will
764 be concatenated to form a new sequential block with
765 that name.

766 The non-unique name of sequential block can only occur
767 at the highest hierarchical level within an analog
768 construct. It is also not allowed to use non-unique
769 names for sequential blocks within a generate construct
770 as that would violate the scope rules for unnamed
771 generate blocks.

772 On reflection, option (b) does not bring any functionality to
773 continued analog constructs that cannot be achieved by

-
- 774 creating a single analog construct in the first place.
- 775 3. As a variation on the above option, require that the statement
- 776 inside the analog construct to continue is a named sequential
- 777 block. A qualifier keyword is now added after the colon
- 778 following the `begin` keyword of the named sequential block, but
- 779 before the actual name of the block. This would concatenate the
- 780 identically named sequential blocks that are marked this way
- 781 inside multiple analog constructs.
- 782 4. Use a concatenate construct at the module level that would
- 783 collect all analog constructs to be concatenated between the
- 784 keywords `concatenate` and `endconcatenate`.
- 785 a. The concatenate construct can only contain analog
- 786 constructs and generate constructs.
- 787 b. Alternatively, the concatenate construct can only contain
- 788 analog constructs and analog generate constructs.
- 789 Analog generate constructs are generate constructs
- 790 whose bodies may only contain analog constructs or
- 791 analog generate constructs.

792 It appears that allowing generate constructs in general inside a

793 concatenate region does not impact non-analog constructs such

794 as structural descriptions or digital behavioral descriptions. In

795 that sense option (a) is the preferred approach as it is

796 essentially simpler and does not require the introduction of the

797 concept of analog generate constructs.

- 798 5. Use a ``default_analog` compiler directive that takes as a
- 799 single argument either the keyword `concatenate` or the
- 800 keyword `concurrent`. This directive can only appear outside
- 801 analog constructs and outside generate constructs. The default
- 802 value of the directive is `concurrent`.

803 All analog constructs appearing in a module body where the

804 ``default_analog` directive has been set to `concatenate`, but

805 before the end of the module body or before the

806 ``default_analog` directive has been set to `concurrent` if that

807 came earlier, will be concatenated together.

808 Options (4) and (5) make the analog construct continuation a

809 rather course-grained selection. Of these two option (4) is

810 preferred as it makes the concatenation process part of the model

811 description instead of focussing on the compiler action alone.

812 Options (1), (2) and (3) differ only in the level at which the

813 concatenation occurs. Option (2) essentially requires that all

814 continued analog constructs should be contained in a sequential

815 block while option (1) does not restrain the syntax in that respect.

816 Option (3) expands on option (2) by requiring that the sequential

817 blocks be named sequential blocks. For the implementation there

818 will probably not be any difference. These three options as opposed

819 to options (4) and (5) do operate only on the analog constructs so

820 conceptually they are clearer than the last two which appear to act

821 also on the non-analog constructs within their region of

822 concatenation.

823 Consequentially, there is a preference for option (1a).

824

4 Examples

825

826

827

828

For the examples we will use a reasonably complex model of an analog-to-digital converter which employs generate constructs with analog constructs inside and will be able to use multiple concurrent analog blocks for modelling various aspects of the design.

829

4.1 Concurrent Analog Constructs

830

831

Here examples are provided to illustrate each of the options proposed in section 3.1.

832

4.1.1 Upward hierarchical referencing

833

834

835

Below is an example of a multiphase oscillator model that uses upward hierarchical referencing as considered in section 3.2.1.1 to refer to the value of the previous phase.

836

Example 6. Multiphase oscillator example

837

838

839

840

841

842

843

844

845

846

847

848

849

850

851

852

853

854

855

856

857

858

859

860

861

862

863

864

865

866

867

```
module multiphase_osc (out, out_q);
  parameter integer stages = 5 from [0:inf];
  parameter real fullscale = 1.0;
  parameter real dly = 0.0;
  parameter real ttime = 10n;
  localparam integer odd = (stages-1)%2.0 from [0:1];
  output [0:stages-1] out, out_q;
  electrical [0:stages-1] out, out_q;

  genvar i;

  generate
    for (i = 0; i < stages; i = i + 1)
      begin : stage
        real value;

        analog begin
          value =
            transition(stage[(i-1)%stages].value,
              dly, ttime);
          V(out[i]) <+ fullscale - value;
          V(out_q[i]) <+ value;
        end
      end
    endgenerate

  analog // Initialization
    @(initial_step)
      stage[0].value = fullscale;

endmodule // multiphase_osc
```

868

4.1.2 Race conditions

869

870

871

872

873

874

There is a possible race condition for initialization in [Example 6 above](#). Even though each variable `value` in the successive iterations of the loop generate construct gets assigned a value in only one particular analog construct, the variable `value` of scope `stage[0]` also gets an assignment in the analog construct at the end of the module.

875

Option (1) of section 3.1.1 would explicitly disallow this use.

876

Option (2) of this section would require the use of a specific

```

877 attribute for this second assignment.
878     analog // Initialization
879     @(initial_step)
880     (* race_condition *)
881     stage[0].value = fullscale;

```

882 A warning would be issued with option (3); to switch off this
883 warning the attribute assignment shown above can be used.

884 Option (4) would not give any warning as the transition statement
885 would assure that the two assignments are not applied at the same
886 step in the analog simulation cycle.

887 4.2 Continued Analog Constructs

888 Here examples are provided for each of the options proposed in
889 section 3.2.

890 4.2.1 Continued analog construct with preceding qualifier

891 This is the adc example of section 2.2, but using the syntax as
892 proposed by option (1a) in section 3.2.2:

893 **Example 7.** ADC example with preceding qualifier.

```

894 module adc (in, out);
895     parameter integer bits = 8 from [0:inf);
896     parameter real fullscale = 1.0;
897     parameter real dly = 0.0;
898     parameter real ttime = 10n;
899     input in;
900     output [0:bits-1] out;
901     electrical in;
902     electrical [0:bits-1] out;
903     localparam real thresh = fullscale/2.0;
904     real sample, value;
905     genvar i;
906
907     analog
908         sample = V(in);
909
910     generate
911         for (i = bits - 1; i >= 0; i = i - 1)
912             concatenate analog begin
913                 if (sample > tresh) begin
914                     value = fullscale;
915                     sample = 2.0 * (sample - thresh);
916                 end
917                 else begin
918                     value = 0.0;
919                     sample = 2.0 * sample;
920                 end
921                 V(out[i]) <+
922                     transition(value, dly, ttime);
923             end
924         endgenerate
925
926     endmodule // adc

```

927 Here the qualifier keyword `concatenate` is applied before the
928 analog construct inside the generate construct. This would
929 concatenate the contents of these analog constructs to the one
930 preceding the generate construct.

931 When the above example is expanded during the elaboration
932 phase would be equivalent to the following code (only using bits =
933 2 to limit the size of the code).

934

Example 8. Expanded ADC example with preceding qualifier.

```
935 module adc (in, out);
936     parameter integer bits = 2 from [0:inf];
937     parameter real fullscale = 1.0;
938     parameter real dly = 0.0;
939     parameter real ttime = 10n;
940     input in;
941     output [0:bits-1] out;
942     electrical in;
943     electrical [0:bits-1] out;
944     localparam real thresh = fullscale/2.0;
945     real sample, value;
946     genvar i;
947
948     analog
949         begin
950             sample = V(in);
951             begin : genblk1[0]
952                 if (sample > tresh) begin
953                     value = fullscale;
954                     sample = 2.0 * (sample - thresh);
955                 end
956                 else begin
957                     value = 0.0;
958                     sample = 2.0 * sample;
959                 end
960                 V(out[0]) <+
961                     transition(value, dly, ttime);
962             end
963             begin : genblk1[1]
964                 if (sample > tresh) begin
965                     value = fullscale;
966                     sample = 2.0 * (sample - thresh);
967                 end
968                 else begin
969                     value = 0.0;
970                     sample = 2.0 * sample;
971                 end
972                 V(out[1]) <+
973                     transition(value, dly, ttime);
974             end
975         end
976
977     endmodule // adc
```

978 The generate construct adds new scopes to the analog sequential
979 block for each iteration of the loop.

980 As an alternative here is the same adc example, now using the
981 syntax as proposed by option (1b):

Example 9. Alternative ADC example with preceding qualifier.

```
982 module adc (in, out);
983     parameter integer bits = 8 from [0:inf];
984     parameter real fullscale = 1.0;
985     parameter real dly = 0.0;
986     parameter real ttime = 10n;
987     input in;
988     output [0:bits-1] out;
989     electrical in;
990     electrical [0:bits-1] out;
991     localparam real thresh = fullscale/2.0;
992     real sample, value;
993     genvar i;
994
995     concatenate analog
996         sample = V(in);
997
998     generate
```

```

000         for (i = bits - 1; i >= 0; i = i - 1)
001             concatenate analog begin
002                 if (sample > tresh) begin
003                     value = fullscale;
004                     sample = 2.0 * (sample - thresh);
005                 end
006                 else begin
007                     value = 0.0;
008                     sample = 2.0 * sample;
009                 end
010                 V(out[i]) <+
011                     transition(value, dly, ttime);
012             end
013         endgenerate
014
015     endmodule // adc

```

016 Here the qualifier keyword **concatenate** is applied before the
017 analog construct inside the generate construct. This would
018 concatenate the contents of all analog constructs preceded by the
019 qualifier until the end of the module or until an analog construct
020 without the preceding qualifier keyword is encountered.

021 4.2.2 Continued analog sequential block

022 This is the adc example of section 2.2, but using the syntax as
023 proposed by option (2) in section 3.2.2:

024 **Example 10.** ADC example with qualified sequential block.

```

025 module adc (in, out);
026     parameter integer bits = 8 from [0:inf);
027     parameter real fullscale = 1.0;
028     parameter real dly = 0.0;
029     parameter real ttime = 10n;
030     input in;
031     output [0:bits-1] out;
032     electrical in;
033     electrical [0:bits-1] out;
034     localparam real thresh = fullscale/2.0;
035     real sample, value;
036     genvar i;
037
038     analog
039         concatenate begin
040             sample = V(in);
041         end
042
043     generate
044         for (i = bits - 1; i >= 0; i = i - 1)
045             analog
046                 concatenate begin
047                     if (sample > tresh) begin
048                         value = fullscale;
049                         sample = 2.0 * (sample - thresh);
050                     end
051                     else begin
052                         value = 0.0;
053                         sample = 2.0 * sample;
054                     end
055                     V(out[i]) <+
056                         transition(value, dly, ttime);
057                 end
058             endgenerate
059
060     endmodule // adc

```

061 Here the qualifier keyword **concatenate** is applied after the begin
062 keyword of an analog sequential block. This would concatenate the

1063 contents of these analog sequential blocks to the one in the analog
 1064 construct preceding the generate construct.
 1065 A slight variation is achieved by using named blocks, an
 1066 considering the `concatenate` keyword on the first occurrence of
 1067 the named block as optional.

1068 **Example 11.** ADC example with qualified named seq. block.

```

1069 module adc (in, out);
1070 parameter integer bits = 8 from [0:inf);
1071 parameter real fullscale = 1.0;
1072 parameter real dly = 0.0;
1073 parameter real ttime = 10n;
1074 input in;
1075 output [0:bits-1] out;
1076 electrical in;
1077 electrical [0:bits-1] out;
1078 localparam real thresh = fullscale/2.0;
1079 genvar i;
1080
1081 analog
1082 begin : body
1083 real sample, value;
1084 sample = V(in);
1085 end
1086
1087 generate
1088 for (i = bits - 1; i >= 0; i = i - 1)
1089 analog
1090 concatenate begin : body
1091 if (sample > tresh) begin
1092 value = fullscale;
1093 sample = 2.0 * (sample - thresh);
1094 end
1095 else begin
1096 value = 0.0;
1097 sample = 2.0 * sample;
1098 end
1099 V(out[i]) <+
1100 transition(value, dly, ttime);
1101 end
1102 endgenerate
1103
1104 endmodule // adc
  
```

1105 Here the qualifier keyword `concatenate` is applied after the `begin`
 1106 keyword of the second (and later) instances of a named analog
 1107 sequential block. This would concatenate the contents of these
 1108 named analog sequential blocks to the one in the analog construct
 1109 preceding the generate construct. The name of the block creates a
 1110 separate scope in which the local variables `sample` and `value` are
 1111 declared.

1112 4.2.3 Continued analog named sequential block

1113 This is the adc example of section 2.2, but using the syntax as
 1114 proposed by option (3) in section 3.2.2:

1115 **Example 12.** ADC example with named sequential block.

```

1116 module adc (in, out);
1117 parameter integer bits = 8 from [0:inf);
1118 parameter real fullscale = 1.0;
1119 parameter real dly = 0.0;
1120 parameter real ttime = 10n;
1121 input in;
1122 output [0:bits-1] out;
  
```

```

123     electrical in;
124     electrical [0:bits-1] out;
125     localparam real thresh = fullscale/2.0;
126     real sample, value;
127     genvar i;
128
129     analog
130     begin : stage
131         sample = V(in);
132     end
133
134     generate
135     for (i = bits - 1; i >= 0; i = i - 1)
136     analog
137     begin : continue stage
138         if (sample > tresh) begin
139             value = fullscale;
140             sample = 2.0 * (sample - thresh);
141         end
142         else begin
143             value = 0.0;
144             sample = 2.0 * sample;
145         end
146         V(out[i]) <+
147             transition(value, dly, ttime);
148     end
149     endgenerate
150
151 endmodule // adc

```

152 Here the keyword **continue** is applied before the name of a named
153 analog sequential block. This would concatenate the contents of
154 these analog sequential blocks to the one in the analog construct
155 preceding the generate construct.

156 4.2.4 Concatenate construct

157 This is the adc example of section 2.2, but using the syntax as
158 proposed by option (4) in section 3.2.2:

159 **Example 13.** ADC example with concatenate construct.

```

160 module adc (in, out);
161     parameter integer bits = 8 from [0:inf);
162     parameter real fullscale = 1.0;
163     parameter real dly = 0.0;
164     parameter real ttime = 10n;
165     input in;
166     output [0:bits-1] out;
167     electrical in;
168     electrical [0:bits-1] out;
169     localparam real thresh = fullscale/2.0;
170     real sample, value;
171     genvar i;
172
173     concatenate
174
175     analog
176     sample = V(in);
177
178     generate
179     for (i = bits - 1; i >= 0; i = i - 1)
180     analog begin
181         if (sample > tresh) begin
182             value = fullscale;
183             sample = 2.0 * (sample - thresh);
184         end
185         else begin
186             value = 0.0;

```

```

1187         sample = 2.0 * sample;
1188     end
1189     V(out[i]) <+
1190         transition(value, dly, ttime);
1191     end
1192 endgenerate
1193
1194     concatenate
1195
1196 endmodule // adc

```

1197 Here a concatenate construct is used to define a region within
1198 which all analog constructs would be concatenated into a single
1199 analog construct.

1200 4.2.5 Concatenate compiler directive

1201 This is the adc example of section 2.2, but using the syntax as
1202 proposed by option (5) in section 3.2.2:

1203 **Example 14.** ADC example with qualified sequential block.

```

1204 module adc (in, out);
1205     parameter integer bits = 8 from [0:inf];
1206     parameter real fullscale = 1.0;
1207     parameter real dly = 0.0;
1208     parameter real ttime = 10n;
1209     input in;
1210     output [0:bits-1] out;
1211     electrical in;
1212     electrical [0:bits-1] out;
1213     localparam real thresh = fullscale/2.0;
1214     real sample, value;
1215     genvar i;
1216
1217     `default_analog concatenate
1218
1219     analog
1220         sample = V(in);
1221
1222     generate
1223         for (i = bits - 1; i >= 0; i = i - 1)
1224             analog begin
1225                 if (sample > thresh) begin
1226                     value = fullscale;
1227                     sample = 2.0 * (sample - thresh);
1228                 end
1229                 else begin
1230                     value = 0.0;
1231                     sample = 2.0 * sample;
1232                 end
1233                 V(out[i]) <+
1234                     transition(value, dly, ttime);
1235             end
1236     endgenerate
1237
1238     `default_analog concurrent
1239
1240 endmodule // adc

```

1241 Using the compiler directive ``default_analog` a region within
1242 which all analog constructs would be concatenated into a single
1243 analog construct. The default concatenation of blocks is switched
1244 on using the `concatenate` value for the ``default_analog`
1245 compiler directive and is switched off using the `concurrent` value
1246 for this directive.