Appendix A
Expression Sampling

In concurrent assertions, in the scope of always_ff procedures in checkers and in several other SVA constructs sampled values of expressions are used. In most cases this sampling is done in the Preponed region, but there are exceptions for different corner cases. In this appendix we provide an exhaustive formal definition of sampling borrowed from the LRM with some modifications.

A.1 Default Sampled Value

As the first step we define the default sampled value—the sampled value that an expression has at the beginning of simulation. In the examples below we will use notation $\Phi$ for the default sampled value; this notation, of course, is not a part of SystemVerilog.

The default sampled value of an expression is defined as follows:

- The default sampled value of a static variable is the value assigned at its declaration.

  \textit{Example A.1.} Given the declaration:
  \begin{verbatim}
  logic [7:0] a = 8'h15;
  \end{verbatim}
  $\Phi(a) = 8'h15$. \hfill $\square$

- If no value is assigned to the static variable at its declaration then the default sampled value is the default value of the corresponding type.

  \textit{Example A.2.} Given the declaration:
  \begin{verbatim}
  logic [7:0] b;
  \end{verbatim}
  $\Phi(a) = 8'hxx$, because the variables of type \texttt{logic} are initialized by default with x. \hfill $\square$
• The default sampled value of any other variable or net is the default value of the corresponding type.

Example A.3. Given the declaration:

```verilog
circuit sv02 {wire w = a;}
```

\( \Phi(a) = 1'b0 \), because the default sampled value of a net is the default value of its type.

• If s is a sequence then the default sampled values of s.triggered and s.matched are false (1'b0).

• The default sampled value of an expression is defined recursively.

Example A.4. Given the declarations:

```verilog
circuit sv02 {bit a; logic [3:0] b = 7, c = 4;}
```

\( \Phi(a || b > c) = \Phi(a) || \Phi(b > c) = 1'b0 || \Phi(b) > \Phi(c) = 7 > 4 = 1'b1 \).

A default sampled value is used in the definition of sampled value functions when there is need to reference a sampled value of an expression before time 0 (see Sect. 7.2.1).

### A.2 Sampled Value of Variable

The general rule for variable sampling is as follows:

• The sampled value of a variable in time slot 0 is its default sampled value defined in Sect. A.1.

• The sampled value of a variable in any other time slot is the value of the variable in the Preponed region of this time slot.

This rule has the following exceptions:

• The sampled value of an automatic variable is its current value. When a past or a future value of an automatic variable is referenced by a sampled value function, the current value of the automatic variable is taken instead. The indication of past or future is thus disregarded.

Example A.5. The sampled value of \( i \) in the following loop:

```verilog
for (int i = 0; i < 7; i++) begin
    ...
end
```

is its current value. If \( a \) is a vector, then in both $sampled(a[i])$ and $past(a[i])$ the current value of \( i \) in the current time slot is assumed.
• The sampled value of a local variable (see Chaps. 15 and 16) is its current value.
• The sampled value of a free checker variable (see Sect. 23.1) is its current value.
If a free checker variable is referred to by a sampled value function (see Sect. 7.2),
then sampling takes place in the Postponed region of the corresponding clock tick.

Example A.6. Given the following declaration:

```vhdl
rand bit v;
$sampled(v)
```

is the current value of \( v \). In the expression \$past(v) the value of
\( v \) is taken from the Postponed region of the previous clock tick. The rationale
of this definition for past or future sampled value functions is to take the final
value of the variable in the corresponding clock tick: the notion of the current
value makes sense for the current clock tick only.

• An input variable of a clocking block must be sampled by the clocking block with

\#1step sampling.\(^1\) This is the sampled value of a such variable (see Sect. 2.3 and
the LRM).

### A.3 Sampled Value of Expression

Having defined the sampled value of a variable, we now can define the sampled
value of an expression recursively:

• The sampled value of an expression consisting of a single variable is the sampled
value of this variable.
• The sampled value of a const cast expression is the current value of its argument.

Example A.7. \$sampled(const'(a)) is the current value of \( a \).

• When a past or a future value of a const cast expression is referenced by a sampled
value function, the current value of this expression is taken instead.

Example A.8. \$past(const'(a)) is the current value of \( a \).

• The sampled value of the sequence methods \( \text{triggered} \) and \( \text{matched} \) is defined
as the current value returned by the sequence method.
• When a past or a future value of a sequence method is referenced by a sampled
value function, this value is sampled in the Postponed region of the corresponding
past or future clock tick

Example A.9. If \( s \) is a sequence then in \$past(s.triggered) the value of
\( s.triggered \) is sampled in the Postponed region of the previous tick of the
corresponding clock. The rationale of this definition is to take the final value

---

\(^1\)The clocking block sampling may be defined other than \#1step, but in this case the input
variable cannot be used in contexts requiring variable sampling in SVA sense, such as a body
of a concurrent assertion.
of \texttt{s.triggered} at a past or future time slot, because the notion of the current value makes sense for the current time slot only. □

- The sampled value of any other expression is defined recursively using the values of its operands.

\textit{Example A.10.} If \texttt{a} is a static variable and \texttt{s} is a sequence then
\[
\text{\$sampled(a \&\& s.triggered) = \$sampled(a)\&\& \$sampled(s.triggered)},
\]
i.e., the value of \texttt{a} is taken from the Preponed region, and the value of \texttt{s.triggered} is the current value. □

\textit{Example A.11.} If \texttt{a} and \texttt{s} are variables and \texttt{f} is a function then
\[
\text{\$sampled(f(a, b)) = f($sampled(a), $sampled(b)).}
\]

\textit{Discussion:} The LRM is not clear whether the global variables accessed by the function are sampled or not. Our interpretation is that these variables should be sampled. Otherwise, it would cause sampling inconsistency when a function is invoked from a right-hand side of a checker NBA. See, for example, function \texttt{next\_tx\_ptr} in Fig. 23.5, Lines 11–18. Note that in the concurrent assertion context functions must be automatic and have no side effects (see Sect. 5.1), therefore the definition is accurate in this case. □
References

22. D. Bustan, Mantis Item 290: Recursive properties can define non-regular languages, Erratum submitted to the IEEE 1800 SV-AC as part of the development and revision of IEEE 1800–2005, November 2004
References

53. The Open Group, Assertion Definition Language (2009), http://adl.opengroup.org/about/index.html
64. F. Vahid, Digital System Design with SystemVerilog, 2nd edn. (Wiley (1000), New York, 2010)


Index

Symbols
import, 42

A
abort, 301, 302, 305
accept on, 227, 283, 288, 302, 305–307, 380, 385, 510
asynchronous, 291, 301, 305–307, 309
nested, 307, 309
reject on, 227, 283, 288, 302, 305, 307, 385, 422
sync accept on, 227, 288, 302, 305, 309, 380, 385
synchronous, 291, 301, 305, 309, 310
sync reject on, 227, 288, 302, 305, 309, 385, 411
abort condition, 283, 288, 291, 305–311, 380, 509
ABV, 14, 18, 87
action block, 63–65, 67–69, 73, 79, 80, 82, 84, 93, 142, 143, 151, 158, 283, 302, 308, 324, 326, 379, 420, 440, 565
action blocks, 159
antecedent, 119
approximation, 456
argument
const ref, 283
ref, 283
assert, 477
assert statement, 477
assertion, 5, 18, 20, 62, 526
assert #0, 62
assert final, 63
assert property, 63
assert, 62
analog, 18
concurrent, 19, 315, 318
procedural concurrent assertion, 315
defered, 15, 18, 19, 28, 65, 66, 70, 71, 142, 162, 167, 172, 176, 315, 492, 561
flush point, 67
report queue, 67
final, 354, 355
defered, 63, 66
immediate, 19, 63, 84, 176, 492, 564, 566
simple, 62, 65, 70
observed
defered, 63, 66
static, 334
assertion control
expect control
runtime violation, 156
assertion coverage, 16
assertion modeling, 189
assertion statement, 20, 61, 73, 82, 91, 151, 322, 388, 413, 419, 498
assertion-based verification, 14
associative array, 354, 356
assume statement, 477
assume-guarantee, 463, 546
assumption, 17, 20, 86–90, 477
assume #0, 87
assume final, 87
assume property, 87
assume, 87
concurrent, 88
defered, 88
immediate, 87
automatic variable, 284, 323, 578

© Springer International Publishing Switzerland 2015
E. Cerny et al., SVA: The Power of Assertions in SystemVerilog,
DOI 10.1007/978-3-319-07139-8
B
bad prefix, 480
bad state, 481
binary relation, 468
black-box verification, 14
BMC, 455
Boolean
  connective, 106
  occurrence, 348

C
Cartesian product, 468
CEX, 454, 461, 462
$\text{changed}$, 150
$\text{changed}_{\text{gclk}}$, 152
characteristic function, 474
checker, 26, 187
  assume set, 544
  free variable, 524
    fully assigned, 534
  instantiation, 205
    procedural, 205
    static, 205
  procedure, 335
  rigid variable, 548
  variable, 211
checker library, 561
  checker-based combinational, 567
  checker-based temporal, 571
  let-based combinational, 566
  property-based temporal, 570
classification, 563
  configurability, 564
  encapsulation, 564
  packaging, 564
  temporality, 564
module-based conversion, 576
clock, 76, 273
  gated, 77, 146
global, 78, see primary clock
$\text{global}_{\text{clock}}$, 78, 154, 472
primary, see global clock, 472
system, see global clock, see primary clock
clock convergence
  continuity, 295
clock domain crossing, 18, 150
clock flow, 280, 286, 287
clock inferencing, 318
clock rewrite rules, 506
clock scoping, 286
clock tick, 273
clocking, 36
  default, 11, 37, 273, 276, 284
reused assertion, 439
default sampled value, 577
delay range, 132
initial, 132
design methodology, 4, 9, 453
disable clause, 302
disable condition, 302
defable iff, 15, 81, 301, 302
default, 304
nesting, 304
defable statement, 332, 333
defable statement, 316
disjunction property, see property operators, or
dist, 89
don't care, 454
DUT, 62

e
elaboration, 32
elaboration time, 32
empty match, 352, 375, 378
empty model, 458, 478
emulation, see hardware acceleration
environment, 62
equivalence verification, 17
evaluation
disabled, 81
evaluation attempt, 74, 345
assertion control tasks, 156
control of, 156
control of action block, 158
efficiency, 445
end time, 75
start time, 75
event, 45
evaluation event, 47
update event, 46
event control, 274, 318
iff, 320
sequence, 265

F
fail action, 63
fairness, 106, 155
false negative, 23, 456
false path elimination, 17
false positive, 456
$fell, 148
$fell_gclk, 152
finite automaton, 469
acceptance, 470
first-order logic, 525
flow diagram, 408
followed by, see property operators, suffix conjunction
for-loop, 324
formal semantics, 495, 511
clocks, 505
resets, 509
formal specification language, 7
formal verification, 5, 23, 355, 470
formal verification flow, 460
formal verification method
complete, 455
incomplete, 455
free variable, 284, 523
function
bit vector, 137
$future_gclk, 152

G
glitch, 49, 303

H
hardware acceleration, 23
high-level model, 9
hybrid verification, 464

I
$inferred_clock, 277
$inferred_clock, 193, 196
$inferred_disable, 193, 196
interface, 38
$isunbounded, 200
$isunknown, 140, 565, 576

K
Kripke structure, see formal verification model

L
language, 469
finitary, 469
infinitary, 469
leading clock, 74, 281, 285
semantic, 287, 289
lemma, 462
let, 24, 167–177
arguments, 169
scoping rules, 169
letter, 469
liveness, 480
  general, 480
local variable, 281, 284, 345, 367, 579
  argument, 359, 367, 371, 400, 403
default actual, 374
direction, 372, 392
assignment, 347, 351, 352, 375
  within repetition, 377
become unassigned, 385
body, 367, 368
color, 512
declaration, 347, 359, 368, 371
declaration assignment, 359, 367, 369, 376
delay, 369
flow, 382, 511
initialization assignment, 369, 376
input, 389
multiplicity of matching, 388
output, 392
receiver, 392
reference, 380
threads, 367, 376, 381
unassigned, 369
logical operator
  unclocked, 295

M
matched, 264, 296, 310, 389, 391, 394
minterm, 474
model, 470
model checking, 480
model language, 473
model relation, 477
multicycle path, 18
multiply clocked, 274, 278

N
negation property, see property operators, not
next state function, 531, 532
next state variable, 474
nexttime, 497

O
$onehot, 24, 139
$onehot0, 87, 139
overapproximation, 456

P
package, 41
pass action, 63
$past, 15, 142, 346
past temporal operators, 261
$past_gclk, 152
PLI, 46
pop_front, 351
procedure, 33
always_comb, 316
always, 316
initial, 316
alwaysalways, 33
always_comb, 33, 34
always_ff, 33, 34
always_latch, 33, 34
initial, 33
structured, 33
program, 36, 40, 265
projection
trace, 526
property, 26, 97, 177–180, 183–184
Boolean, 98, 496
hybrid, 480
liveness, 104
mutually recursive, 399, 402
negation, 503
next occurrence, 248
recursive, 399
restrictions, 411
safety, 480
sequential, 113
strong, 487
weak, 487
property operators
  bounded always, 241
  bounded eventually, 240
property operators
  s_until_with, 499
  until_with, 499
property coverage, 92
property operators, 228
always, 101, 236
  implicit, 102
and, 230, 497
Boolean connectives, 229
bounded s_always, 241
bounded s_eventually, 241
case, 233
if, 232
if-else, 232
iff, 231, 498
implies, 230, 498
nexttime, 100, 238
not, 230, 497
or, 230, 498, 500
sequence property, 228
s_eventually, 103, 236, 499
s_nexttime, 238
strong sequence, 228
suffix conjunction, 234
non-overlapping, 234
overlapping, 234
suffix implication, 119, 234, 504, 505
non-overlapping, 25, 119, 234
overlapping, 27, 119, 234
s_until, 236, 497
s_until_with, 237
until, 108, 236, 499
until_with, 108, 237
weak sequence, 228
protocol
FIFO, 349, 358, 403
pipeline, 346
rety, 405
sequential, 347, 379, 405
tag, 353, 360
pruning, 459
free, 459
set, 459
PSL, 27, 75, 248, 550
push_back, 351
quantifier
existential, 469
universal, 469
queue, 350
procedural assertion, 284
quiescent point, 21
race, 50
region
Active, 47, 50
Inactive, 47
NBA, 47
Observed, 50, 51
Postponed, 50, 51
Preponed, 50, 51, 274, 283
Re-Inactive, 51
Re-NBA, 51
Reactive, 50, 51, 283
region set
Active, 47
reactive, 51
relation
total, 470
repetition range, 127
infinite, 128
reset, 301, 302
asynchronous, 302
default, 11
synchronous, 302
reset condition, 301
general, 310
restriction, 90
$rose, 148
$rose_gclk, 152
RT, 17
RTL, 3, 5, 9, 13–15, 17, 18, 24, 70, 87, 187, 209, 211, 356, 420, 439, 453, 460, 462, 470, 472, 481, 482, 487, 523, 540, 561
Rule of Clock Inference, 319
safety property, 480
$sampled, 141, 283, 349
sampled value function, 140, 475
global clocking, 152
future, 152
past, 152
sampling, 20, 22, 79, 274, 283, 577
satisfiability, 478
sequence, 25, 111, 177–183
Boolean, 112, 500
bounded, 113, 135
conjunction, 253
disjunction, 500
empty, 125, 500
iteration, 501
match, 111
empty, 114, 130
method, 257–265
multiply clocked, 286
unbounded, 135
sequence coverage, 92
sequence match item, 378
sequence method, 296, 391
sequence operators
intersect, 251, 500
and, 253
concatenation, 115, 500
consecutive repetition, 124
disjunction, see or
first_match, 256, 501
sequence operators (cont.)
- fusion, 117, 500
- goto repetition, 247
- initial delay, 118
- nonconsecutive repetition, 250
- or, 126, 500
- throughout, 246, 253
- within, 255
- zero repetition, 125

sequence property, see sequential property

Short-circuiting, 66, 72

simulation, 21
- glitch, 65, 66, 69, 76, 79, 81, 88
- random, 3, 22, 89, 94, 465

simulation semantics, 45

simulation time, 32

singly clocked, 273

SoC, 79

SSA, see static single assignment, 538

$stable, 86, 150

$stable_gclk, 152

starvation, 106

state, 469
- accepting, 469
- initial, 469

statement
- wait, 266

static single assignment, 538

static variable, 322

subroutine
- attached to sequence, 362, 378

subsequence
- maximal singly clocked, 286

SVTB, 3

synchronizer, 278
- unclocked, 295

synthesis, 9, 318

SystemC, 9, 13

T
- tight satisfaction, see sequence, match, 500
- time slot, 55
- timing verification, 17
- TLA, 13
- trace, 89, 97, see word
- transaction, 21
  - pending, 21
- transition relation, 469, 531, 538
- triggered, 257, 282, 296, 310, 362, 363, 389, 391, 394

U
- unclocked semantics, 506
- underapproximation, 456, 457

V
- vacuity, 123, 458
- vacuous evaluation, 242
- vacuous execution
  - rules of nonvacuity, 242
  - vacuous success, 242
- validation
  - post-silicon, 18
- validity, 477
- verification bound, 455

Verilog, 3

VPI, 46

W
- white-box verification, 15
- word, 469
  - empty, 469
  - finite, 469
  - infinite, 469