SystemVerilog Assertions (SVA)

Ming-Hwa Wang, Ph.D.
COEN 207 SoC (System-on-Chip) Verification
Department of Computer Engineering
Santa Clara University

Introduction

- Assertions are primarily used to validate the behavior of a design
- Piece of verification code that monitors a design implementation for compliance with the specifications
- Directive to a verification tool that the tool should attempt to prove/assume/count a given property using formal methods
- Capture the design intent more formally and find specification error earlier
- Find more bugs and source of the bugs faster
- Encourage measurement of function coverage and assertion coverage
- Re-use checks throughout life-cycle, strength regression testing

Formal Method

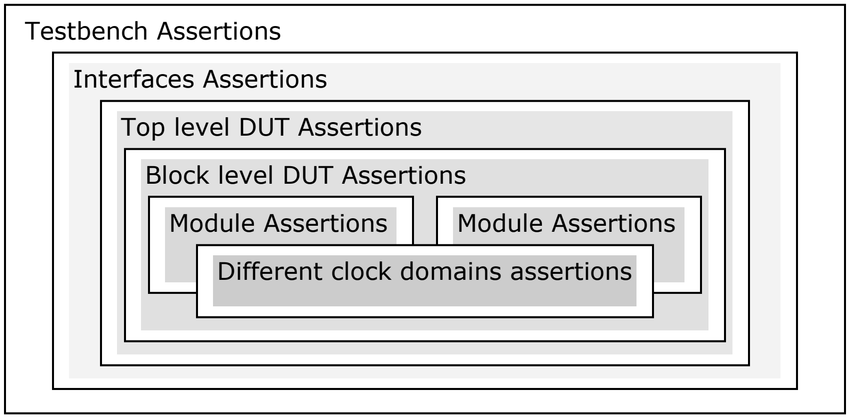
Formal assertion-based verification flow

Benefits of Assertions

- Improves observability of the design
  - Using assertions one can create unlimited number of observation points any where in the design
  - Enables internal state, datapath and error pre-condition coverage analysis
- Improves debugging of the design
  - Assertion help capture the improper functionality of the DUT at or near the source of the problem thereby reducing the debug time
  - With failure of assertion one can debug by considering only the dependent signals or auxiliary code associated to the specific assertion in question
  - Assertion also helps to capture bugs, which do not propagate to the output
- Improves the documentation of the Design
  - Assertions capture the specification of the Design. The spec is translated into an executable form in the form of assertions, assumptions, constraints, restrictions. The specifications are checked during the entire development and validation process
  - Assumptions in assertions capturing the design assumptions continuously verify whether the assumptions hold true throughout the simulation
  - Assertions always capture the specification in concise form which is not ambiguous i.e., assertions are the testable form of requirements
  - Assertions go along with the design and can also be enabled at SOC level

- Assertion can be used to provide functional coverage
  - Functional coverage is provided by cover property
  - Cover property is to monitor the property evaluation for functional coverage. It covers the properties/sequences that we have specified
  - We can monitor whether a particular verification node is exercised or not as per the specification
  - Can be written for
    - Low-level functionality coverage inside a block
    - High-level functionality coverage at interface level
- Can use these assertions in formal analysis
  - Formal analysis uses sophisticated algorithms to prove or disprove that a design behaves as desired for all the possible operating states. One limitation is that it is effective only in block level not at full chip or SOC level
  - Desire behavior is not expressed in a traditional test bench, but rather as a set of assertions. Formal analysis does not require test vectors
  - With Formal analysis many bugs can be found quickly and very easily in the Design process without the need to develop large sets of test vectors

Where SVA can reside?



Who writes Assertions?

- White-Box Verification
  - Inserted by design engineers
  - Block Interfaces
  - Internal signals
- Black-box Verification
  - Inserted by
    - IP Providers
    - Verification Engineers
  - External interfaces
  - End-to-end properties

Different Assertion Languages

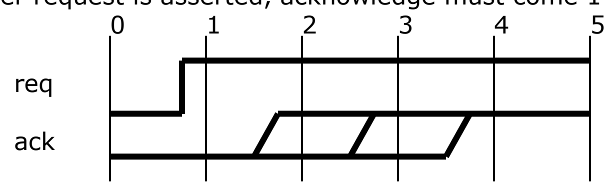
- PSL (Property Specification Language) – based on IBM Sugar
- Synopsys OVA (Open Vera Assertions) and OVL (Open Vera Library)
- Assertions in Specman
- 0-In (0–In Assertions)
- SystemC Verification (SCV)
- SVA (SystemVerilog Assertions)

Why SVA?

- SystemVerilog – a combination of Verilog, Vera, Assertion, VHDL – merges the benefits of all these languages for design and verification
- SystemVerilog assertions are built natively within the design and verification framework, unlike a separate verification language
- Simple hookup and understanding of assertions based design and test bench – no special interfaces required
- Less assertion code and easy to learn
- Ability to interact with C and Verilog functions
- Avoid mismatches between simulations and formal evaluations because of clearly defined scheduling semantics
- Assertion co-simulation overhead can be reduced by coding assertions intelligently in SVA

SystemVerilog Assertion Example

A concise description of complex behaviour:
After request is asserted, acknowledge must come 1 to 3 cycles later



assert property(@(posedge clk**) $rose(**req**) |-> ##[**1:3**] $rose(**ack**));**

Properties and Assertions

Types of SVA

- Immediate Assertions
- Concurrent Assertions

Immediate Assertions

- Immediate assertions = instructions to a simulator
- Follows simulations event semantics
- Appears as a procedural statement, executed like a statement in a procedural block
- Syntax: **assert (**expression**)** pass_statement [**else** fail_statement]
- The statement is non-temporal and treated as a condition in if statement
- The else block is optional, however it allows registering severity of assertion failure

- Severity System tasks:
  - **$fatal** : run time fatal, terminates simulation
  - **$error** : run time error (default)
  - **$warning** : run time warning, can be suppressed by command-line option
  - **$info** : failure carries no specific severity, can be suppressed
- All severity system tasks print the severity level, the file name and line number, the hierarchical name or scope, simulation time, etc.
- Example:

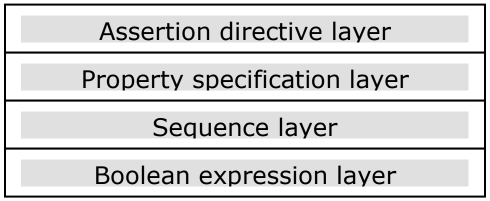
```
always @ (posedge clk) begin:checkResults
   assert ( output == expected ) okCount++;
   else begin
         $error(“Output is incorrect”);
         errCount++;
   end
end
```

Concurrent Assertions

- Concurrent assertions = instructions to verification tools
- Based on clock semantics. Evaluated on the clock edge
- Values of the variables used in evaluation are the sampled values
- Detects behavior over a period of time
- Ability to specify behavior over time. So these are called temporal expressions
- Assertions occur both in procedural block and a module
- Example:

```
assert property (
   @(posedge clk) a ##1 b |-> d ##1 e
);
```

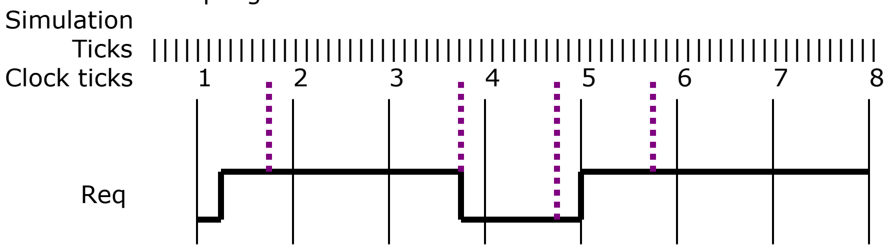
- Layers of Concurrent Assertion
  - Make the sequence
  - Evaluate the sequence
  - Define a property for sequence with pass fail
  - Property asserted with a specific block (eg: Illegal sequence, measuring coverage …)



  - Boolean expression layer
    - Elementary layer of Concurrent assertion
    - Evaluates Boolean expression to be either TRUE or FALSE
    - Occur in the following of concurrent properties
      - In the Sequences used to build properties
      - In top level disable iff claues

assert property (@(posedge clk**) disable iff (**a **&&** **$rose(**b, **posedge** clk**)) trigger |=>** test_expr**;**

- restrictions on the type of variables shortreal, real and realtime
  - string
  - event
  - chandle
  - class
  - associative array
  - dynamic array
- Functions in expressions should be automatic
- Variable in expression bust be static design variable
- Sampling a variable in concurrent assertions



- The value of signal req is low at clocks 1. At clock tick 2, the value is sampled as high and remains high until clock tick 4. The sampled value req at clock tick 4 is low and remains low until clock tick 6
- Notice that, at clock tick 5, the simulation value transitions to high. However, the sampled value is low

- Sequence layer: build on top of Boolean expression layer, and describe sequence made of series of events and other sequences
  - Linear sequence: absolute timing relation is known
  - Nonlinear sequence
    - multiple events trigger a sequence and not time dependant
    - multiple sequences interact with and control one another
  - Sequence block
    - Define one or more sequences
    - Syntax:

      sequence identifier **(**formal_argument_list**);**
      variable declarations
      sequence_spec
      endsequence

    - Example:

```
sequence seq1           sequence seq2          sequence seq3
    ~reset##5 req;          req##2 ack;            seq1##2 ack
Endsequence             endsequence            endsequence
```

  - Usage: sequence can be instantiated in any of the following blocks
    - A module
    - An interface block
    - A program block
    - A clocking block
    - A package
    - A compilation unit scope
- ## delay operator: used to join expression consisting of events.
  - Usage:
    - **##** integral_number
    - **##** identifier
    - **## (**constant_expression**)**
    - **## [**cycle_delay_const_range_expression**]**
  - The operator ## can be used multiple times within the same chain. E.g., a ##1 b ##2 c ##3 d
  - You can indefinitely increase the length of a chain of events using ## and 1'b1. The example below extends the previous chain of events by 50 clocks. E.g., a ##1 b ##2 c ##3 d ##50 1'b1
  - Sequence overlap indicates b starts on the same clock when a ends: a ##0 b
  - Sequence concatenation means b starts one clock after a ends: a ##1 b
  - You can use an integer variable in place of the delay. E.g., a ##delay b
  - The following means b completes 2 clock ticks after a completes (regardless of when b starts): a ##2 b.ended
  - You can specify a range of absolute delays too. E.g., a ##[1:4] b. You can also use a range of variable delays. E.g., a ##[delay1:delay2] b
  - The symbol $ in a delay range indicates that a signal or event will 'eventually' occur. E.g., a ##[delay1:$] b
- Sequence and clock
  - Implied clock

```
sequence seq1
    ~reset##5 req;
endsequence
```

  - Using clock inside a sequence

```
sequence Sequence3;
    @(posedge clk_1) // clock name is clk_1
    s1 ##2 s2; // two sequences
endsequence
```

- Sequence operations

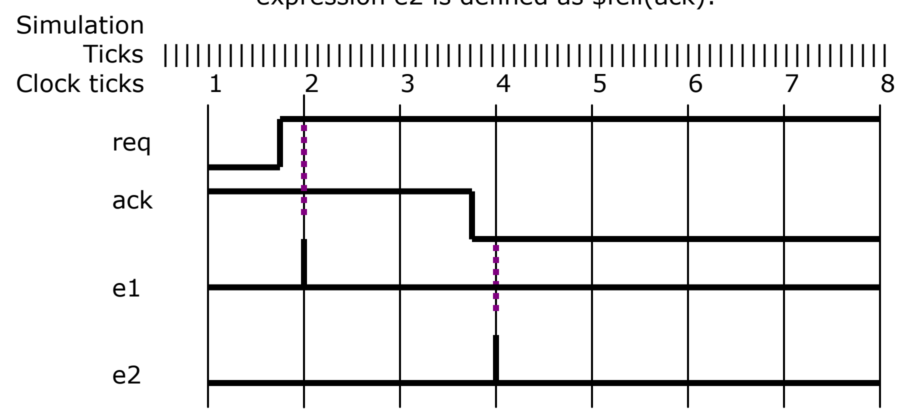
Category	Operators	Associativity
repetition	[*] [=] [->]	-
cycle delay	##	left
match	throughout, within, intersect, and , or	right for throughout, left for others

  - Repetition operators
    - There are three types of repetition operators.
      - Consecutive Repetition Operator [*]
      - Non-consecutive Repetition Operator [=]
      - Goto Repetition Operator [->]

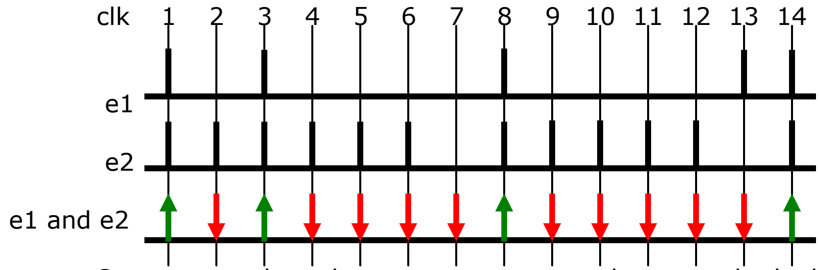
- Consecutive repetition operator
    - Indicates that the sequence repeats itself a specified number of times. E.g., s1 ##2 s2 [*4] ##5 s3 is same as s1 ##2 (s2 ##1 s2 ##1 s2 ##1 s2) ##5 s3 or, or simply s1 ##2 s2 ##1 s2 ##1 s2 ##1 s2 ##5 s3
    - Empty Sequence [*0]: a repetition of 0 times indicates that the resultant is empty
        - Usage rules
            - Neither *(e ##0 s)* nor *(s ##0 e)* matches any sequence.
            - *(e ##n s)* is equivalent to *(##(n-1) s)*, if n > 0
            - *(s ##n e)* is equivalent to *(s ##(n-1) `true)*, if n > 0
    - Repetition with a Range
        - Range can be specified with repetition operator. E.g., s1 [*2:3] is equivalent to s1 ##1 s1 (two times of s1) or s1 ##1 s1 ##1 s1 (three times of s1)
        - A range repetition is applicable to a chain of sequences (or events) as well. E.g., (s1 ##5 s2) [*2:3] is equivalent to (s1 ##5 s2) ##1 (s1 ##5 s2) (two times of (s1 #5 s2)) or (s1 ##5 s2) ##1 (s1 ##5 s2) ##1 (s1 ##5 s2) (three times of (s1 #5 s2))
        - An upper bound '$' in a range indicates the sequence indicates specified lower bound. E.g.,
            - s1[*2:$]
            - s0 ##3 s1[*2:$] ##2 s2
- Non-Consecutive exact repetition operator of Boolean expression, extends beyond true value of operand to last true value
    - *b [=3]*: The Boolean expression *b* has been true thrice, but not necessarily on successive clocks and there may be additional clock cycles after the last true *b* before the sequence completes.
    - *b [=3:5]*: Here, *b* has been true 3, 4 or 5 times, once again not necessarily on consecutive clocks, and with possible additional clocks afterwards when *b* is not true.
    - *a ##2 b [=3] ##4 c*: The Boolean expression *b* has been true thrice, but not necessarily on successive clocks. The first occurrence of *b* happens after two clocks cycles of a. The last one occurs *at least* four clock cycles before c.
- Goto Repetition Operator
    - Goto Repetition operator of Boolean expression, end at true value of expression
        - *b [->3]*: The Boolean expression *b* has been true thrice, but not necessarily on successive clocks
        - *b [->3:5]*: Here, *b* has been true 3, 4 or 5 times, once again not necessarily on consecutive clocks
        - *a ##2 b [->3] ##4 c*: The Boolean expression *b* has been true thrice, but not necessarily on successive clocks. The first occurrence of *b* happens after two clocks cycles of a. The last one occurs four clock cycles before c
- Value change functions: SVA sample value functions detect events on signals/expressions and can be used within assertions

Function	Meaning
$rose (expression) e.g. : *(a ##1 b) ##1 $rose(c)*	true, if the least significant bit of the expression changed to 1; false, otherwise
$fell (expression) e.g. : *(a ##1 b) ##1 $fell(c)*	true, if the least significant bit of the expression changed to 0; false, otherwise
$stable (expression) e.g. : *(a ##1 b) ##1 $stable(c)*	true, if the value of the expression did not change; false, otherwise
$past (expression, number_of_ticks) e.g. : a == $past(c, 5)	returns the sampled value of the expression that was present number_of_ticks prior to the time of evaluation of $past.

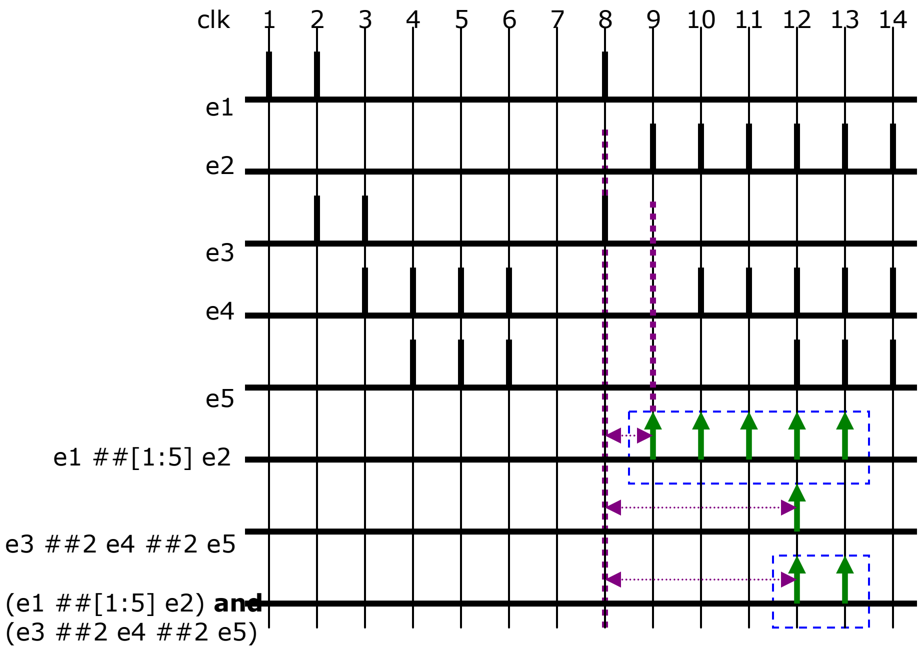
- Value change expression example: value change expression e1 is defined as $rose(req) and value change expression e2 is defined as $fell(ack):



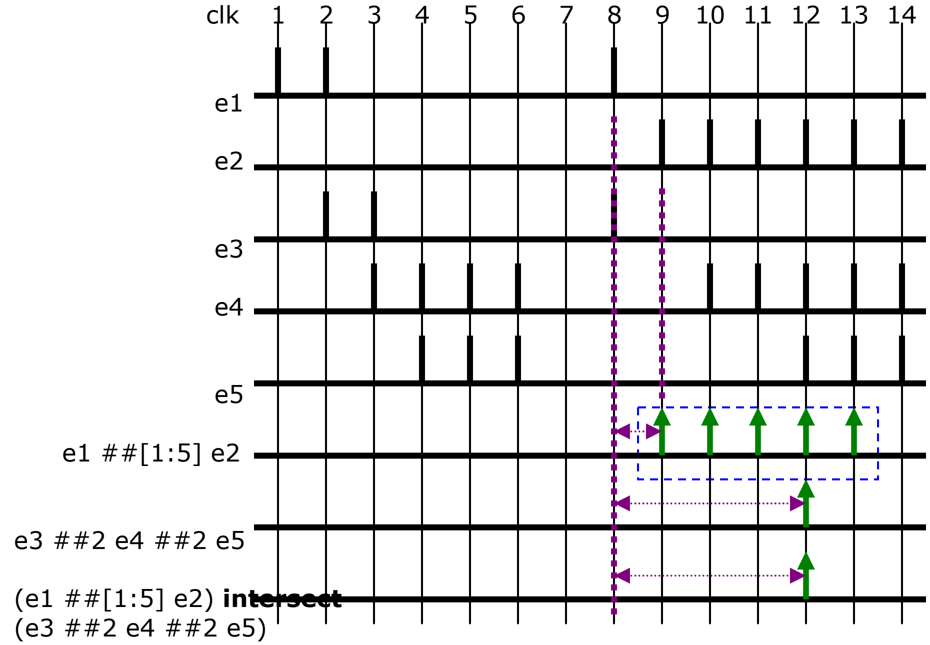
- Boolean and: and two Booleans



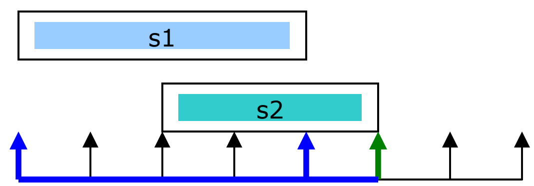
- Sequence and: and two sequences – and expects both the events to match but end time can be different



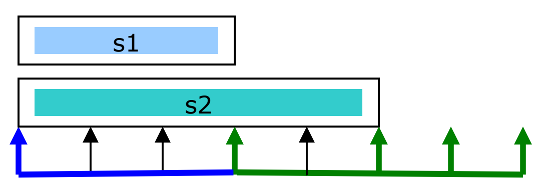
- Intersect construct: intersecting two sequences – intersect expects both the events to match but end time must be same



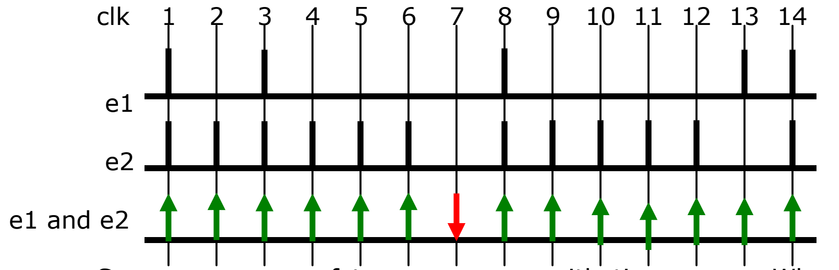
- Sequence ended: the ended method returns a Boolean that is true in the cycle at which the associated sequence has achieves a match, regardless of when the sequence started. E.g., s1 ##1 s2.ended



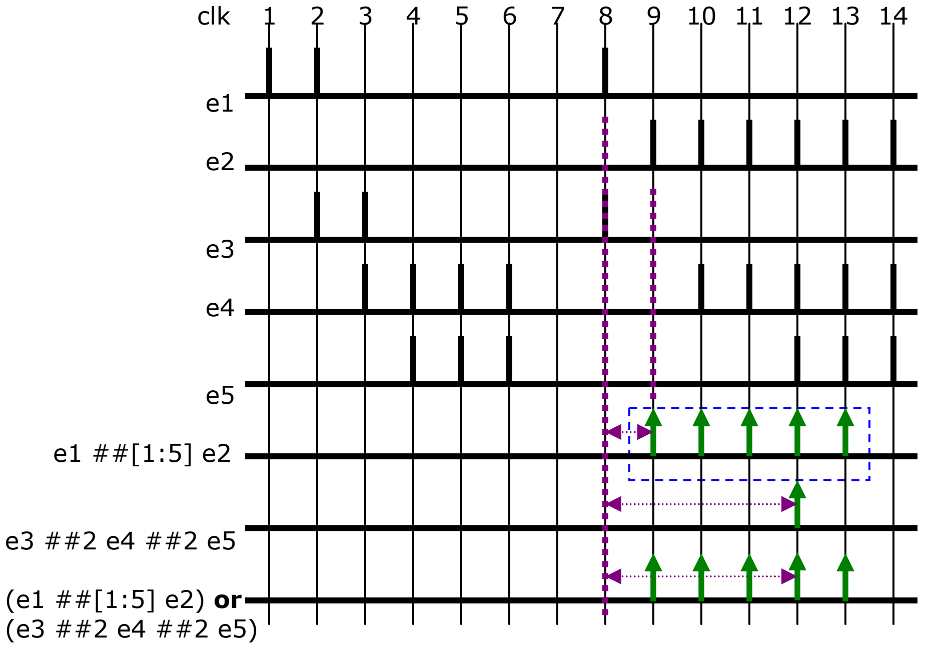
- Sequence "or": the Sequence s1 and s2 has multiple matches – when s1 matches and each of the samples on which s2 matches. If s1 matches, "or" sequence also matches, regardless of whether s2 matches and vice versa. E.g., s1 or s2



- Boolean or: or two Booleans

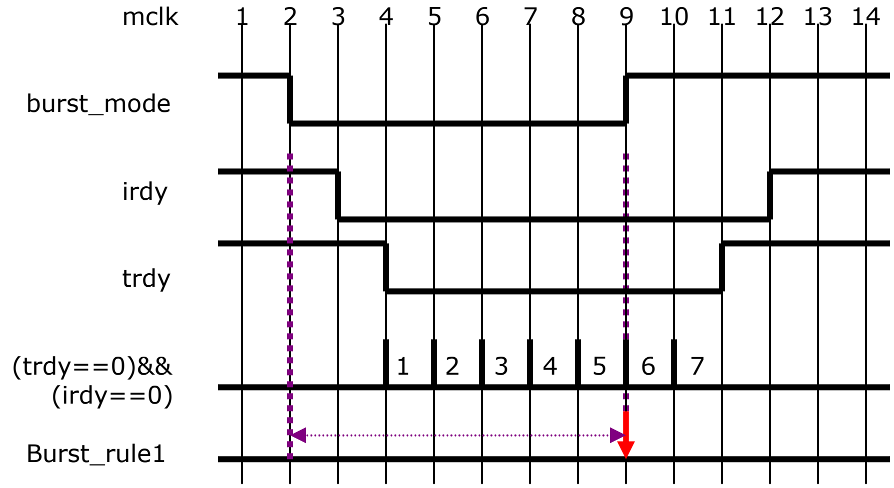


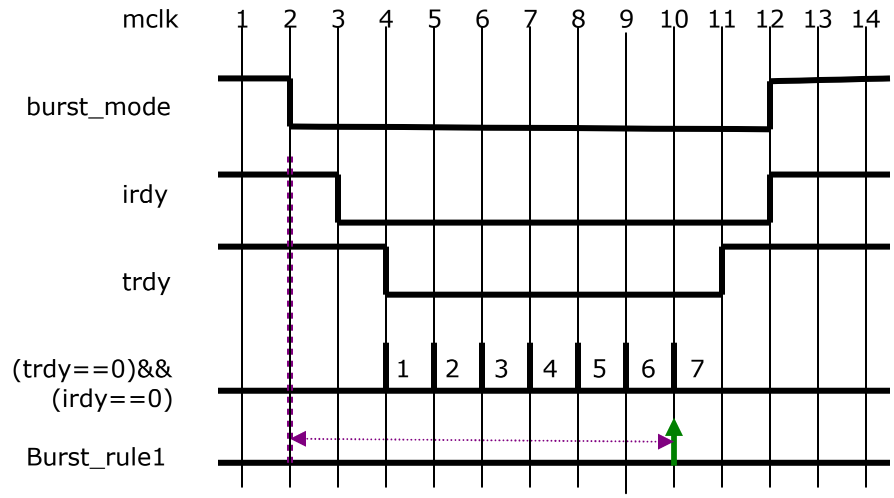
- Sequence or: or of two sequences with time range. When first_match will be asserted in this sequence?



- Throughout construct example

```
sequence burst_rule1;
    @(posedge mclk)
        $fell(burst_mode) ##0
        (!burst_mode) throughout (##2 ((trdy==0)&&(irdy==0)) [*7]);
endsequence
```





- Local variables
  - Sequences Can have local variable
  - New copy of local variable is created and no hierarchical access to these local variables are allowed
  - Assigned using a comma separated list along with other expression
  - Eg:

```
sequence s1( int test );
    int i;
    i = test;
    (data_valid, (i=tag_in)) |-> ##7 (tag_out==i);
endsequence
```

  - Cannot be accessed outside the sequence where it is instantiated

        - Local variable can be passed only as an entire actual argument
    - System functions

Function	Meaning
$onehot(expression)	true, if only one of the bits in the expression is high
$onehot0(expression)	true, if *at most* one of the bit in the expression is high
$isunknown(expression)	true, if any bit of the expression is X or Z
$countones(expression)	returns the number of 1s in a bit vector expression

- Property layer
    - Built on the foundation of Sequences, Boolean expressions
    - Property block

        property identifier **(**formal_arg_list**);**
            variable declaration
            property_spec
        endproperty

    - Property declaration can occur in
        - A module
        - An interface
        - A program
        - A clocking block
        - A package
        - A compilation unit
    - Property declaration does not affect a simulation behavior until the property is designated as following
        - An assumed or anticipated behavior: By associating the property using an *assume* keyword. The verification environment assumes that the behavior occurs
        - A checker: By associating the property using an *assert* keyword. The verification environment checks if the behavior occurs
        - A coverage specification: By associating the property using a *cover* keyword. The verification environment uses the statement for measuring coverage
    - Types of Properties
        - Property Type 1: A Sequence
            - A property expression may be a simple sequence expression as shown below

```
property sequence_example;
    s1; // s1 is a sequence defined elsewhere
endproperty
```

            - A sequence as a property expression is valid if the sequence is not an empty match (i.e., it contains a specific non-empty expression to match).
        - Property Type 2: Another Property
            - An instance of a named property can be used as a valid property expression. For instance, the property *sequence_example* defined above is itself can be a property expression

```
property property_example;
    Sequence_example
endproperty
```

            - A property may call itself resulting in a recursive property
        - Property Type 3: Property Type Inverse
            - A property expression may be an inverse of another property expression. The inversion is done by using the *not* operator

```
property inversion_example;
    not Sequence_example
endproperty
```

        - Property Type 4: Property Type Disjunction
            - A disjunction property is true if either of its constituent property expressions is true. The disjunction operator *or* is used to describe a disjunction operator

```
property disjunction_example;
    sequence_example or inversion_example;
endproperty
```

        - Property Type 5: Property Type Conjunction
            - A conjunction is equivalent of a logical and operation, and very aptly, is expressed by an *and* operator

```
property conjunction_example;
    sequence_example and inversion_example
endproperty
```

        - Property Type 6: An 'if..else'
            - An 'if...else' property expression is a conditional statement that describes two possible behaviors based on the value of an expression

```
property ifelse_example;
    if ( expr == 2'b10)
        inversion_example;
    else  sequence_example
endproperty
```

        - Property Type 7: An Implication
            - An implication property describes a behavior that occurs when a preceding behavior takes place
            - The implication operators '|->' and '|=>' are used for describing such a property

```
property conjunction_example;
    s0 |-> sequence_example
endproperty
```

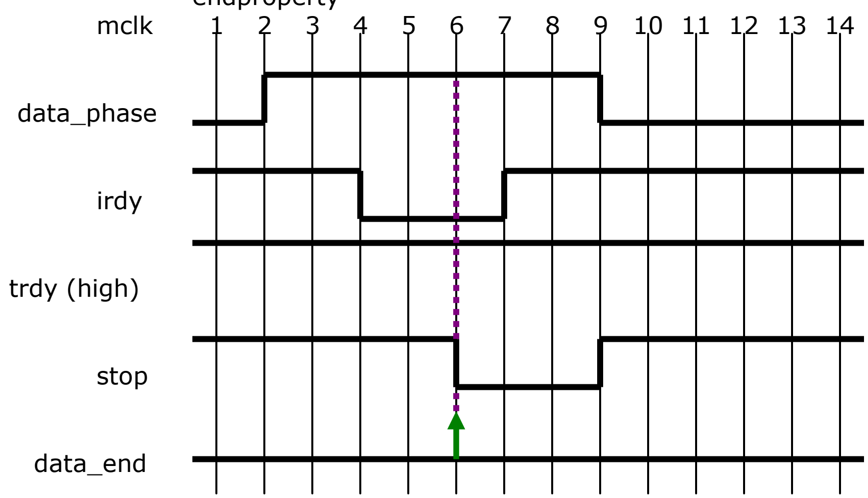
- Implication construct
    - Two Types
        - ->

- =>
- Usage:
  Res = A (Antecedent) -> B (Consequent)
- Note:
  - Antecedent seq_expr can have multiple success
  - If there is no match of the antecedent seq_expr, implication succeeds vacuously by returning true.
- Truth table for Res is:

A	B	Res
0	0	Vacuous success
0	1	Vacuous success
1	0	False
1	1	True

- Example

```
Property data_end;
    @(posedge mclk)
    Data_phase |->
        ((irdy==0)&&($fell(trdy) || $fell(stop)));
endproperty
```



- Implication construct

Sequence Operator	Property Operator	Associativity
[*], [=], [->]		-
##		left
throughout		right
within		left
intersect		left
	not	-
and	and	left
or	or	left
	if..else	right
	|->, |=>	right

- Concurrent Assertion using sequence

```
sequence s1;
   @(posedge clk)  a ##1 b ##[1:2] c;
endsequence;
My_Assertion : assert property (@(posedge clk) s1);
```

- Concurrent Assertion using property

```
property p1;
   @(posedge clk) s1 ##1 s1 ##1 s1;
endproperty
Top_Assertion : assert property (p1) pass_stmt;
      else fail_stmt;
```

- Also have “cover” construct. Can be used for functional coverage.
  cover property (p1);
- Property expression qualifiers
  - Clocking event
    - The clocking event describes when a property expression should take place. An example of this is shown below.

```
property clocking_example;
    @(posedge clk) Sequence_example
endproperty
```

  - Disable iff
    - A 'disable iff' command is similar to a reset statement - the property expression is valid only if the reset situation is lifted. Here is an example of this command

```
property disable_iff_example;
    Disable iff (reset_expr) Sequence_example
endproperty
```

- Recursive properties
  - Eg:

```
property recursive_always;
    Sig_x and (1'b1 |=> recursive_always);
endproperty
```

  - Restrictions for Recursive Properties
    - A recursive property can not use a *not* operator.
    - The operator *disable iff* can not be used in a recursive property.
    - A recursive property can call itself only after a positive time delay (to avoid an infinite loop).
- Assertion layer
  - Adds sense to the property described
  - Key words that define a sense for a assertion
    - assert: The keyword *assert* indicates that a property acts a checker. The verification environment should check if the behavior occurs.
    - assume: The *assume* keyword indicates that the property behavior is anticipated or assumed and should be treated so by the verification tool.

- cover: If a property is associated with the keyword *cover*, it indicates that the property evaluation will be monitored for coverage.
- Concurrent Can be specified inside the following construct
  - an always block
  - an initial block
  - a module
  - a program
  - an interface
- When instantiated outside the scope of a procedural block (initial or always), a property behaves as if it is within an always block.

```
assert property (p1);
```

- outside the scope of a procedural block is equivalent to: always

```
assert property (p1);
```

- Assert statement
  - Property associated with a assert statement is treated as checker

```
property top_prop;
    seq0 |-> prop0
endproperty
assert to_prop:
assert property (top_prop) begin
    int pass count;
    $display ( "pass: top_prop");
    pass_count = pass_count +1'b1;
end
```

- Assume statement
  - A property associated with an *assume* statement implies that the property holds during verification
  - For a formal or dynamic simulation environment, the statement is simply assumed to be true and rest of the statements that need to be verified are constrained accordingly

```
Assume_property_reset_seq0: assume property (reset_seq0);
property reset_seq0;
    @(posedge clk) reset |-> not seq0;
end
```

- Cover statement
  - A *cover* statement measures the coverage of the various components

```
cover_property_top_prop:
cover property (top_prop)
$display ("top_prop is a hit");
property top_prop;
    seq0 |-> prop0;
endproperty
```

- Expect statement
  - An *expect* statement is very similar to an *assert* statement, but it must occur within a procedural block (including initial or always blocks, tasks and functions), and is used to block the execution until the property succeeds.

```
task mytask;
    ...
    if (expr1)
        expect (my_property)
        pass_block();
    else // associated with the 'expect',
         // not with the 'if'
        fail_block();
    ...
endtask
```

Binding properties to scopes or instances

- To facilitate verification separate from design, it is possible to specify properties and bind them to specific modules or instances.
- Uses:
  - It allows verification engineers to verify with minimum changes to the design code/files.
  - It allows a convenient mechanism to attach VIP to a module or instance.
  - No semantic changes to the assertions are introduces due to this feature. It is equivalent to writing properties external to a module, using hierarchical path name.
  - Example of binding two modules.

```
module cpu ( a, b, c)
    < RTL Code >
endmodule
module cpu_props ( a, b, c)
    < Assertion Properties >
endmodule
```

  - bind cpu cpu_props cpu_rules_1(a, b, c);
    - cpu and cpu_props are the module name.
    - cpu_rules_1 is cpu_props instance name.
    - Ports (a, b, c) gets bound to the signals (a, b, c) of the module cpu.
    - every instance of cpu gets the properties.