Embedded Assertions

```
arbiter(clk, rst, req0, req1, grant0, grant1)
    …
    always @(posedge clk) begin
    if(grant0 & grant1) $display("hello world");
    …
   // this is a bad situation…

endmodule
```

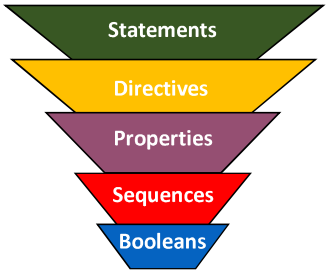
Is there an error if "hello world" is displayed?

- Assertion constructs provide structure to communicate intent and/or energize comments
- Recommendations
  - **Keep it simple**: very complex, confusing assertions can be written, but most useful assertions can be expressed simply
  - **Reference modeling**: often clearer (or necessary!) to write auxiliary code and relate it to the actual RTL using assertions

Statements added to RTL describing behavior expectations
"Comments" evaluated by simulation and formal tools

System Verilog Assertions

assertions ---- assumptions ---- cover points
immediate --- concurrent



Statement Layer:	packaging, SV language constructs
Directives Layer:	intention of property
Property Layer:	implications of sequences
Sequence Layer:	set of Booleans over time
Boolean Layer:	simple expressions

Special thanks to Erik Seligman for examples. Some examples also from Mentor Graphics

Statements and Directives

- Directive types
  - Assert

```
name1: assert           …
name2: assert property …
```

  - Assume

```
name3: assume           …
name4: assume property …
```

  - Cover

```
name5: cover            …
name6: cover  property …
```


Immediate vs Concurrent

Immediate assertions (Boolean only)

- Usable in arbitrary procedures, functions
- Evaluated when reached in code

```
A1: assert (foo && bar || baz);
```

Concurrent assertions

- Usable outside procedures
- Evaluated each clock

```
A2: assert property (foo && bar || baz);

A3: assert property (@(posedge clk) disable iff (rst)
                     (foo ##1 bar |=> baz));
```

clocks and reset

sequence

- Immediate/concurrent assertions can be assumptions or properties

```
M1: assume property (@(posedge clk) disable iff (rst)
                                     (foo ##1 bar |=> baz));
C1: cover property (@(posedge clk) disable iff (rst)
                                    (foo ##1 bar ##1 baz));
```

Boolean Examples


```
always @(*) begin
  a1: assert (foo && bar || baz);
end

always @(posedge clk) begin
   a2: assert property (foo && bar || baz);
end

default clocking @(posedge clk2); endclocking;
a3: assert property (foo && bar || baz);
```


Sequences

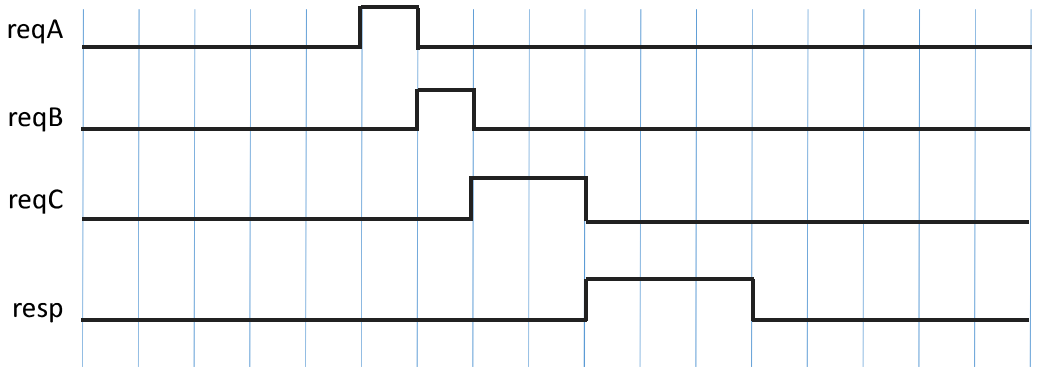
- Basic sequence operations

`s1 ## s2`	`s1` followed by `s2` in the next cycle
`s1 ##n s2`	`s1` followed by `s2` `n` cycles later
`s1 ##[m:n] s2`	`s2` delayed `m` to `n` cycles (0 = overlap)
`s1[*m]` or `[*m:n]`	consecutive repeat `m` times to `n` times
`s1[=m]` or `[=m:n]`	non-consecutive repetition
`s1[->m]` or `[->m:n]`	goto non-consecutive exact repetition
`s1 or s2`	one sequence is true
`s1 and s2`	same start, both eventually true
`s1 intersect s2`	both end on same cycle
`bool` `throughout s1`	`bool` true for all of `s1`

- Usable only in concurrent assertion
- Inherit clock from usage
  - Default clocking / procedure clock
  - Clock of property that uses the sequence

Sequence Examples

`foo ##1 bar ##1 baz`	*foo,* then *bar*, then *baz*
`(foo ##1 bar) or (foo ##1 baz)`	*foo*, then either *bar* or *baz*
`(foo ##1 bar)[*5] ##1 baz`	*foo bar* 5 times, then *baz*
`foo throughout(bar[*5] ##1 baz)`	*foo* = 1 during (bar 5 times and then baz)



Does the above wave form satisfy the following sequences?

```
reqA[*1:2] ##1 reqB
reqA[*1:2] ##1 reqC
reqC[*1:2] ##1 resp
```

Other Useful Building Blocks

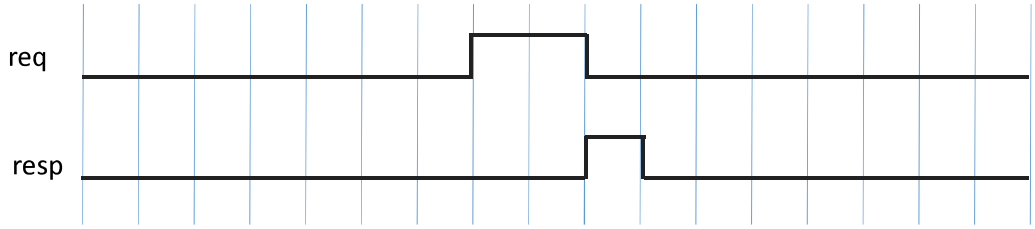
- In bounds, '$' = infinity

```
foo[*3:$] ##1 bar
```

- Special system functions

```
$rose(sig)          $past(sig [,num_of ticks])
$fell(sig)          $stable(sig)
```

Consider: req ##2 resp



Asserting a Sequence

- What does this mean?

```
assert property (foo ##1 bar);
```

  - Checks the sequence *every cycle*
    - Usually not useful!
    - Also expensive in simulation
- Sequences mainly to help build properties
  - OK to use a sequence in a cover property
- Negated sequence is very useful

```
assert property (not(foo ##1 bar));
```

Properties

- Basic operation: triggered (guarded) implication

```
seq |-> prop        overlapping
seq |=> prop        non-overlapping
                    seq |-> ##1 prop
```

- Trigger must be a sequence
  - Corresponds to intuition
  - Easy to check in simulation

Property Examples

|->	overlapping
|=>	non-overlapping

```
foo |=> bar
```

- If we see *foo*,
  then we see *bar* the next cycle

```
foo ##1 bar |-> baz
```

- If we see *foo* and then *bar*,
  then we also see *baz* (same cycle as *bar*)

```
foo [*5] |-> not (bar ##1 baz)
```

- If we see *foo* 5 times,
  then if *bar* is true during the 5th *foo*, *baz* will be false next cycle

```
foo ##0 bar |-> $rose(baz)
```

- If *foo* and *bar* are true at the same time,
  then *baz* must have just risen

Named Properties

- Similar to named sequences

```
property p1(e1,e2);
  e1 |=> e2;
endproperty
```

- The following are equivalent:
  - `assert property (p1(foo,bar));`
  - `assert property (foo |=> bar);`

Named Sequences

```
sequence s1(a);
   a ##1 foo;
endsequence
```

- The following are then equivalent:

```
a1:assert property(s1(bar) |-> baz);
a2:assert property((bar ##1 foo) |-> baz));
```

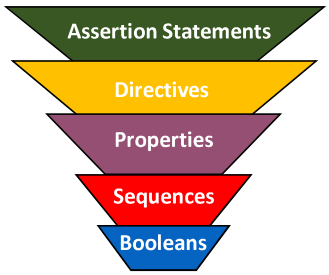
Property syntax

- SVA syntax is fussy
  - Legal: *assert property (foo |=> bar);*
  - Illegal: *assert property foo |=> bar;*
- `AND, OR, NOT` available
  - Don't confuse with boolean operators: `&&, ||, !`

Some examples:

```
assert property (a |-> b) AND (a |-> c);
assert property (NOT (a ##1 b));
assert property (p1(a,b) OR (a |-> b));
```


SVA Recap



Statements:	packaging, SV language syntax
Directives:	intention of property
Property Layer:	implications of sequences
Sequence Layer:	set of Booleans over time
Boolean Layer:	simple expressions

Sequences

```
s1 ## s2
s1 ##n s2
s1 ##[m:n] s2

s1[*m]      or [*m:n]

s1[=m]      or [=m:n]
s1[->m]     or [->m:n]

s1 or s2
s1 and s2
bool throughout s1
```

- '$' = infinity

```
$rose(sig)
$fell(sig)

$past(sig [,number_of ticks])
$stable(sig)
```

A few examples of sequences

cat ## dog	
cat ##0 dog	*really fast dog*
cat ##3 dog	*slower dog*
cat ## [1:3] dog	*dog maybe briefly distracted*
cat ## [3:$] dog	*slower dog might take a nap*
cat ## dog[*2]	*cat followed by 2 dogs*
cat ## dog[*2:5]	*cat followed by 2-5 dogs*

Sequence matching

- Sequence *matched* when all specified values have occurred

cat[*3:4] ## dog

1	2	3	4	5
cat	cat	cat	cat	-
-	-	-	dog	dog

match at cycle 4 and 5

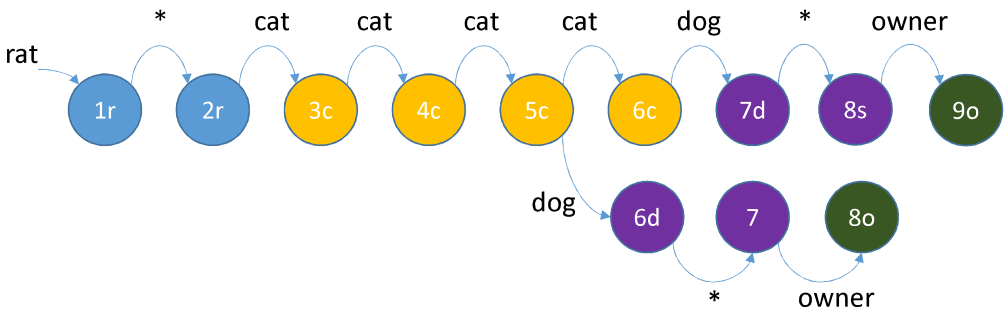
rat ##2 cat[*3:4] ## dog

1	2	3	4	5	6	7	8	9
rat	rat	-	-	-	-	-	-	rat
-	cat	cat	cat	cat	-	-	-	cat
-	-	-	-	dog	dog	-	-	-

match at cycle 6

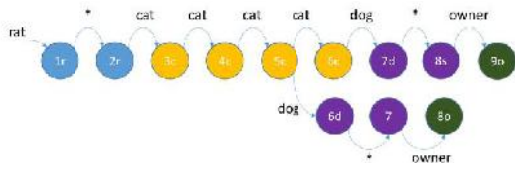
Sequencing automaton

rat ##2 cat[*3:4] ## dog ##2 owner

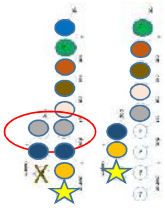


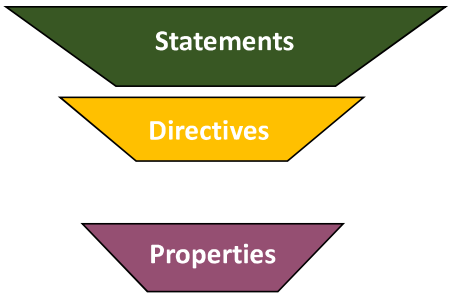
Sequencing automaton

rat ##2 cat[*3:4] ## dog ##2 owner



1	2	3	4	5	6	7	8	9	
rat	rat	-	-	-	-	-	-	-	-
	-	cat	cat	cat	cat	-	-	-	-
	-	-	-	-	dog	dog	-	owner	-






```
name1: assert
name3: assume
name5: cover
name2: assert property
name4: assume property
name6: cover  property
```

```
seq |-> prop
seq |=> prop
```

```
A3: assert property (@(posedge clk) disable iff (rst)…

property p1(e1,e2);
       e1 |=> e2;
endproperty
```


```
$onehot ( expr )
$onehot0 ( expr )
$isunknown (expr )
$countones ( expr )
```

Other Useful Building Blocks

- Disabling properties

```
A3: assert property (@(posedge clk) disable iff (rst)
```

- System functions

`$onehot` (*expr*)
  - true if exactly one bit of the expression is high

`$onehot0` (*expr*)
  - true if at most one bit of the expression is high

`$isunknown` (*expr*)
  - true if any bit of expression is (4 state logic) X or Z

`$countones` (*expr*)
  - counts the number of bits set in a bit vector

Immediate Assertions

```
always @(a or b)
   a1:  assert (a==b);
always @(a)
   b = a;
```

- What is the order of execution for the always blocks?
  - Can a1 be evaluated twice?

- Yes, immediate assertions are glitchy!
  - Order of blocks not defined in Verilog / SV
  - Deferred assertion:

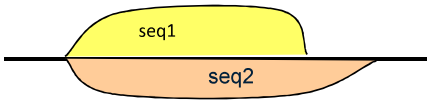
```
assert #0(a==b);
```

➔ Don't use immediate assertions unless really needed!
  - If you have clock, use it:

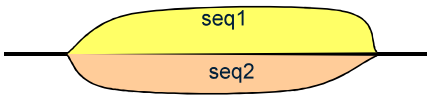
```
A1:assert property (@(posedge clk)(a==b));
```

AND vs INTERSECT

- Two similar sequence ops
- AND = same start



- INTERSECT = same start and end



Sequences vs Properties

- A sequence can be used as a property… BUT
  - The property is the "sequence matched starting every cycle"
  - Don't confuse `a ##1 b` (continuously checked)
    with `a |=> b` (only checked when a is true)
    - Exception: **`initial`** block
- Is the following property useful?

```
assert property @(posedge clk) (foo ##1 bar);
```

foo is always true, and so is **bar** starting on cycle 2

- Negated sequences are properties, not sequences

"This sequence is never matched"

Be Careful With $stable, etc

$stable, $changed, $past use previous trace values

- But what is "previous" at start of sim?
  - Default value often *X*
- So what does this property do?

```
wire foo;
A1: assert property ($stable(foo))
```

A1 claims `foo` is always *X*!

Rethink property, add delay or reset

```
A2: assert property (##1 $stable(foo));
```


Triggered Implication

- How to read a |-> b?
  - "a implies b"? Not exactly
  - "a triggers b"? Better!
- **Some consequences of this definition**
  - Left side must be a sequence, not a property
  - Negated sequence cannot be the left side
    - (negated sequence is property)
  - If want negated sequence to trigger property, need to rethink

```
assert property (!s1) |-> p1;    Illegal
assert property (p1 or s1);      OK… but different
```

Negating Properties

- What does *not (a |-> b)* mean?
  - Doesn't mean: b never happens when a does
  - Does mean: sometimes a happens and b doesn't

- Followed-by (#-#) operator
  - not(a |-> b) rewritten as a #-# not b
  - Read as "at some point a is followed by not b".

Action blocks

- Can specify actions to occur if an assertion passes or fails

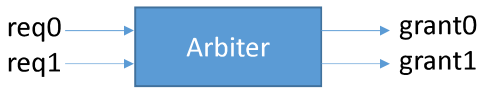
```
assert property( @(posedge clk) disable iff(reset)
    (foo |-> bar)
else begin
     $error("Foo without Bar");
end
```

Make Assertions Part of the Design Process

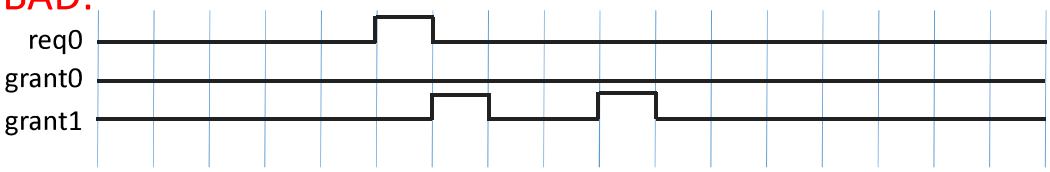
- Define standard assertion note
  // Assertion a123: Check for legal grants;
- Designer adds: spec, testplans, RTL
  - Assertion idea != interrupt thought flow
  - OK to add note if no time to write assertion
- Assertion expert role
  - Scripts to collect assertion notes
  - Help designer implement/focus
- Assertions: casual & easy
  - Pitfall: Treat as "out-of-band" process
  - Pitfall: Avoid requirements seen as penalty
    - "Must eventually prove X% formally"

Write an SVA specifying an arbiter is fair

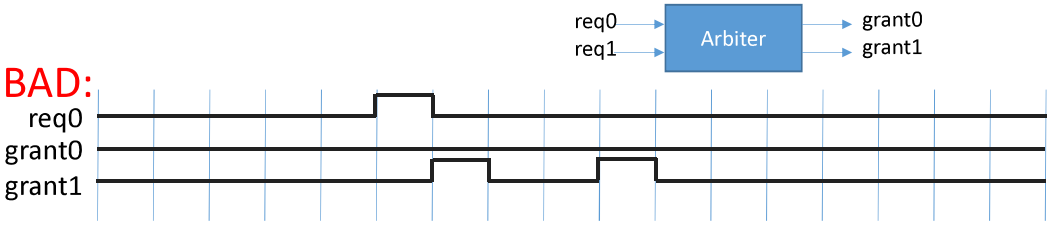
- 2 requests, 2 grants
- requirement: requester should not wait more than two arbitration cycles.



BAD:



Requester should not wait more than two arbitration cycles.



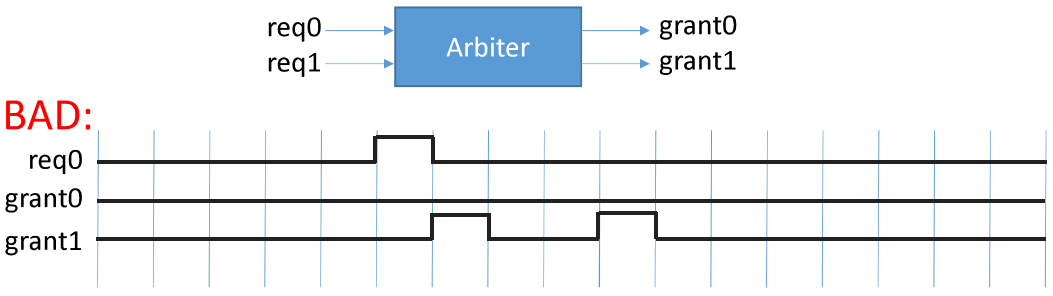

```
s1 ## s2
s1 ##n s2
s1 ##[m:n] s2

s1[*m]      or [*m:n]

s1[=m]      or [=m:n]
s1[->m]     or [->m:n]

s1 or s2
s1 and s2
bool throughout s1
```

Sequences

```
$past(sig [,number_of ticks])
$rose(sig)
$fell(sig)
$stable(sig)
```

Write an SVA asserting an arbiter is fair

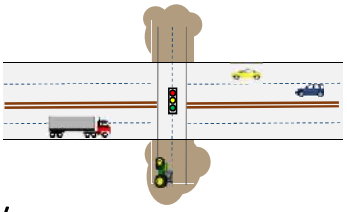
- 2 requests, 2 grants
- requirement: requester should not wait more than two arbitration cycles.



fair_req0: assert property(@(posedge clk) disable iff (reset)

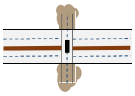
not($rose(req0) ## (!gnt0) throughout (gnt1[->2])))

FarmRoad Traffic Light Controller



- Two roads: farm road and multi-lane highway
- Highway should have right of way
  - Controller must maximize the time the green light remains on
- Outputs:
  - *Long*: Green light must be at the green state for at least *long* time units
  - *Short*: At end of a green period, if there is a car waiting at the farm intersection, the light turns yellow for a short period and then red
- The light remains red for the highway until all cars have crossed the highway or until a *long* timer has expired.

Assignment: write traffic light assertions



- The traffic light on both sides should never be green at the same time
- The short timer should be smaller than the longer timer value
- If there are no more cars on the farm road when the farm light is green, the light should switch to yellow
- It is not possible for a car on the farm road to wait forever

Assigment: (previous exam question)

- Consider a hardware system that supports a single interrupt. When a new interrupt request is made, the signal *iReq* will be set to 1 and when the interrupt is serviced, the signal *serviced*, will be set to 1.

  Write an SVA property that states that whenever an interrupt request is made, eventually the interrupt is serviced and then within 4 clock ticks, the interrupt request signal is set to 0. Assume the system has one clock, *clk* and the property is only valid if the reset signal, *rst* remains 0.