Controlling Constraints

- Constraints can be turned on/off during runtime.
- constraint_mode() is used to achieve this capability.
- When used with handle.constraint, this method controls a single constraint.
- When used with just handle, it controls all constraints for an object.
- To turn off constraint, 0 is passed as an argument to constraint_mode and to turn on, 1 is passed as an argument.

Example

```
class Packet;
rand int length;
constraint c_short { length inside { [1:32] }; }
constraint c_long { length inside { [1000:1023]};}
endclass
```

Example

```
Packet p;
initial begin
p = new;
// Create a long packet by disabling short constraint
p.c_short.constraint_mode(0);
assert (p.randomize());
// Create a short packet by disabling all constraints
// then enabling only the short constraint
p.constraint_mode(0);
p.c_short.constraint_mode(1);
assert (p.randomize());
end
```

Inline Constraints

- New constraints can be added to existing constraints while calling randomize function using randomize with.

- constraint_mode can be used disable any conflicting constraints.

```
class Transaction;
rand bit [31:0] addr, data;
constraint c1 { addr inside { [0:100], [1000:2000] }; }
endclass
```

Inline Constraints

```
Transaction t;
initial begin
t = new();              // addr is 50-100, 1000-1500, data < 10
repeat(5)
assert(t.randomize() with { addr >= 50; addr <= 1500; data < 10;} );

repeat(5)               // force addr to a specific value, data > 10
assert(t.randomize() with { addr == 2000;  data > 10; } );
end
```

Constraint in Inheritance

- Additional constraints can be provided in a subclass class(child class). Child class object has to fulfill both the constraints (parent and child).

```
class base;
rand int data;
constraint limit1 { data> 0;
                    data< 100; }
endclass
```

```
class child extends base;
constraint limit2 { data > 50; }
endclass
```

Parent: 0 < data < 100

Child: 50 < data < 100

Example2

```
class base;
rand int data;
constraint limit1 { data> 0;
                    data< 100;}
endclass
```

```
class child extends base;
constraint limit2 { data == 50;}
endclass
```

Parent: 0 < data < 100

Child: data=50

Example3

```
class base;
rand int data;
constraint limit1 { data> 40;
                    data< 50;
                  }
endclass
```

```
class child extends base;
constraint limit2 { data > 10;
                     data< 30;
                    }
endclass
```

Parent: 40 < data < 50

Child: 10 < data < 30

Randomization Fails because both constraints are not satisfied

Example 4

```
class base;
rand int data;
constraint limit1 { data> 40;
                    data< 50;
                  }
endclass
```

Parent: 40 < data < 50

```
class child extends base;
rand int data;
constraint limit2 { data > 10;
                    data< 30;
                  }
endclass
```

Child: 10 < data < 30

Parent data is different as compared to child data. Data Overridden

Constraint in Inheritance

- Constraints are overridden in child class if they are defined with same name as that present in parent class.

```
class base;
rand int data;
constraint limit { data> 20;
                   data< 40; }
endclass
```

```
class child extends base;
constraint limit { data > 50;
                   data < 90; }
endclass
```

Parent: 20 < data < 40

Child: 50 < data < 90

Constraints are overridden

- Create a class apb with following properties and methods:
- Pdata – 32 bit
- Paddr – 8bit
- Pen – 1 bit
- Pwr_rd- 1 bit

Write a constraint data should be even when pen is high

Try to solve before construct to above constraint

Write a constraint paddr always less 25 when pwr_rd is low

Try to implement rand_mode and constraint_mode

Write a inline constraint for pen will always be low

System Verilog

PROCESSES

final block

- final block is a procedural statement that occurs at the end of simulation.
- Delays are not allowed inside final block.
- Final block can only occur once during simulation.
- $finish can be used to trigger final block.

```
final
begin
$display("Simulation ends");
$display("No of errors=%0d", error);
end
```

Block Statements

- Block statements are used to group procedural statements together.
- There are two types of blocks :
  - Sequential blocks (begin - end)
  - Parallel blocks (fork - join)
- System Verilog introduces three types of Parallel blocks:
  - fork – join
  - fork – join_any
  - fork – join_none

fork - join

- In fork-join, the process executing the fork statement is blocked until the termination of all forked processes.

```
module test;
initial begin $display("Before Fork");
fork
begin #3 $display("#3 occurs at %0d", $time); end
begin #6 $display("#6 occurs at %0d", $time); end
begin #8 $display("#8 occurs at %0d", $time); end
begin #5 $display("#5 occurs at %0d", $time); end
join
$display("Out of Fork at %0d", $time); end
endmodule
```

fork - join

Result :

Before Fork
#3 occurs at 3
#5 occurs at 5
#6 occurs at 6
#8 occurs at 8
Out of Fork at 8

fork – join_any

- In fork-join_any, the process executing the fork statement is blocked until any one of the processes spawned by fork completes.

```
module test;
initial begin $display("Before Fork");
fork
begin #3 $display("#3 occurs at %0d", $time); end
begin #6 $display("#6 occurs at %0d", $time); end
begin #8 $display("#8 occurs at %0d", $time); end
begin #5 $display("#5 occurs at %0d", $time); end
join _any
$display("Out of Fork at %0d", $time); end
endmodule
```

fork – join_any

Result :

Before Fork
#3 occurs at 3
Out of Fork at 3
#5 occurs at 5
#6 occurs at 6
#8 occurs at 8

fork – join_none

- In fork-join_none, the process executing the fork statement continues to execute with all processes spawned by fork.

```
module test;
initial begin $display("Before Fork");
fork
begin #3 $display("#3 occurs at %0d", $time); end
begin #6 $display("#6 occurs at %0d", $time); end
begin #8 $display("#8 occurs at %0d", $time); end
begin #5 $display("#5 occurs at %0d", $time); end
join_none
$display("Out of Fork at %0d", $time); end
endmodule
```

fork – join_none

Result :

Before Fork
Out of Fork at 0
#3 occurs at 3
#5 occurs at 5
#6 occurs at 6
#8 occurs at 8

wait fork

```
program test;
initial begin $display(“Before Fork”);
fork
begin #3 $display(“#3 occurs at %0d”, $time); end
begin #6 $display(“#6 occurs at %0d”, $time); end
begin #8 $display(“#8 occurs at %0d”, $time); end
begin #5 $display(“#5 occurs at %0d”, $time); end
join_none
$display(“Out of Fork at %0d”, $time); end
endprogram
```

program block exits simulation once it reaches end of initial block

wait fork

Result :

Before Fork
Out of Fork at 0

wait fork

```
program test;
initial begin $display(“Before Fork”);
fork
begin #3 $display(“#3 occurs at %0d”, $time); end
begin #6 $display(“#6 occurs at %0d”, $time); end
begin #8 $display(“#8 occurs at %0d”, $time); end
begin #5 $display(“#5 occurs at %0d”, $time); end
join_none
$display(“Out of Fork at %0d”, $time);
wait fork; end
endprogram
```

Waits till all forked processed are completed

wait fork

Result :

Before Fork
Out of Fork at 0
#3 occurs at 3
#5 occurs at 5
#6 occurs at 6
#8 occurs at 8

disable fork

```
module test;
initial begin $display(“Before Fork”);
fork
begin #3 $display(“#3 occurs at %0d”, $time); end
begin #6 $display(“#6 occurs at %0d”, $time); end
begin #8 $display(“#8 occurs at %0d”, $time); end
begin #5 $display(“#5 occurs at %0d”, $time); end
join_any
$display(“Out of Fork at %0d”, $time);
disable fork; end
endmodule
```

Disable fork kills all forked processes

disable fork

Result :

Before Fork
#3 occurs at 3
Out of Fork at 3

always_comb

- System Verilog provides always_comb procedure for modeling combinational logic behavior.

```
always_comb
c= a & b;
```

- There is an inferred sensitivity list.
- The variables written on the left-hand side of assignments shall not be written to by any other process.
- The procedure is automatically triggered once at time zero, after all initial and always procedures.
- Software tools will perform additional check to warn if behavior within always_comb does not match a combinational logic.

always_latch

- System Verilog provides always_latch procedure for modeling latched logic behavior.

```
always_latch
if(en) b=a;
```

- This construct is identical to always_comb, except that the tools will perform additionsal check to warn if behavior does not match a latch logic.

always_ff

- always_ff procedure can be used to model synthesizable sequential logic behavior.

```
always_ff @ (posedge clk iff !rst or posedge rst)
if(rst)
q<=0;
else
q<=d;
```

- The always_ff procedure imposes the restriction that it contains one and only one event control and no blocking timing controls.

- Tools should perform additional checks to warn if the behavior does not represent sequential logic.

Conditional Event Control

- @ event control can have an iff qualifier.
- event expression only triggers if the expression after the iff is true.

```
always @(a iff en==1)
begin
y<= a;
end
```

```
always @(posedge clk iff en)
begin
y<=din;
end
```

Both the event expression (@a and (@posedge clk)) occurs only if en==1

Sequence Event Control

- A sequence instance can be used in event expressions to control the execution of procedural statements based on the successful match of the sequence.

```
sequence abc;
@ (posedge clk) a ##1 b ##1 c;
endsequence
```

```
always @(abc)
$display("event occurred on a, b and c in order");
```

Named Events

- System Verilog allows used to define events and trigger them.

- There are two ways to trigger an event
  - Blocking (->)
  - Non-Blocking(->>)

- There are two ways to wait for an event
  - @ (event_name)
  - wait(even_name.triggered)

Example1

```
event myevent;        //User defined event
int count=0;

initial
begin
-> myevent;           //Triggering event
#3 -> myevent;
end

always @(myevent)   //waiting for event
count+=1;
```

Result :
count=2

Example2

```
event myevent;        //User defined event
int count=0;

initial
begin
-> myevent;          //Triggering event
@(myevent)           //waiting for event
count+=1;
end
```

Result :
count=0

while using @, waiting should start before event is triggered

Example3

```
event myevent;        //User defined event
int count=0;

initial
begin
@(myevent)            //waiting for event
-> myevent;           //Triggering event
count+=1;
end
```

Result :
count=0

@ is waiting for event but event is never triggered.

Example4

```
event myevent;        //User defined event
int count=0;

initial
begin
->>myevent;          //Triggering event in NBA region
@(myevent)           //waiting for event
count+=1;
end
```

Result :
count=1

Event is scheduled to triggered in NBA region because of which waiting starts before triggering and count increments

Example5

```
event myevent;        //User defined event
int count=0;

initial
fork
->myevent;           //Triggering event
@(myevent)           //waiting for event
count+=1;
join
```

count value depends upon which statement is executed first
result varies from simulator to simulator.

Example6

```
event myevent;        //User defined event
int count=0;

initial
begin
->myevent;                    //Triggering event
wait (myevent.triggered)      //waiting for event
count+=1;
end
```

Result :
count=1

When using .triggered, waiting should start before or at same time when event is triggered.

Example7

```
event myevent;        //User defined event
int count=0;

initial
fork
->myevent;                    //Triggering event
wait(myevent.triggered)       //waiting for event
count+=1;
join
```

Result :
count=1

System Verilog

INTER PROCESS COMMUNICATION

Semaphore

- Semaphore is a built-in class which conceptually is a bucket.
- When semaphore is allocated, then a bucket containing fixed number of keys is created.
- Process using semaphore must first procure a key from bucket before they can continue to execute.
- Once process is over, key should be returned back to the bucket.
- Semaphore is basically used to control access to shared resources.

Semaphore - Methods

- new() method is used to create semaphore with specified number of keys. Default key count is 0.
- put() method is used to return specified number of keys to semaphore. Default value is 1.
- get() method is used to procure specified number of keys from semaphore. Default value is 1.

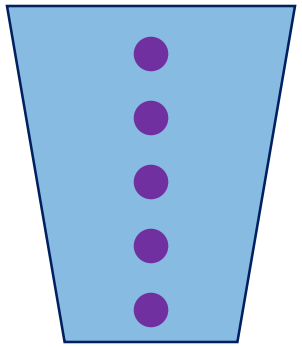
Semaphore - Methods

- In get() method if the specified number of keys is not available, the process blocks until the keys become available.
- try_get() method is used to procure a specified number of keys from a semaphore, but without blocking.
- In try_get() method if the specified number of keys are not available, the method returns 0 else a positive value and continues.

Example

```
semaphore smp;
int got=0;

initial begin
smp=new(5);
#5 smp.get(3);
#6 smp.get(1); got=got +1;
#2 if(smp.try_get(3)) got=got +1;
end
```



```
initial
begin
#8 smp.get(2);
#7 smp.put(2);
end
```

got=1 at 15

Example

```
module test;
semaphore smp;
int a=0;
smp=new(1);

initial
fork
//statement1
//statement2
join
```

```
begin : statement1
smp.get;
a=7;
#3 smp.put;
end statement1

begin : statement2
smp.get;
a=3;
#2 smp.put;
end statement2
```

Mailbox

- Mailbox is a built-in class that allows messages to be exchanged between processes.
- Data can be sent to mailbox by one process and retrieved by another.
- Mailbox can be bounded or unbounded queues.
- Mailbox can be parameterized or Non-parameterized.
- Non-Parameterized mailboxes are typeless , that is single mailbox can send and receive different type of data.

Mailbox - Methods

- new() method is used to create mailbox with size specified as an argument.
- If size is defined as 0 (default) then mailbox is unbound.
- num() method is returns the number of message currently present inside mailbox.
- put() method places a message in a mailbox in a FIFO order.
- If the mailbox is bounded, the process shall be suspended until there is enough room in the queue.

Mailbox - Methods

- try_put() method attempts to place a message in mailbox. This method is meaningful only for bounded mailboxes.
- If mailbox is full this method returns 0 and message is not placed else it returns 1 and message is placed.
- get() method retrieves a message from a mailbox.
- This method removes message from a mailbox and calling process is blocked if mailbox is empty.
- try_get() method attempts to retrieves a message from a mailbox without blocking.

Mailbox - Methods

- peek() method copies message from a mailbox without removing message from the queue.
- If mailbox is empty then current process is blocked till a message is placed in the mailbox.
- If the type of the message variable is not equivalent to the type of the message in the mailbox, a run-time error is generated.
- try_peek() method attempts to copy a message from a mailbox without blocking. If the mailbox is empty, then the method returns 0 else if variable type is different it returns negative number else positive number is returned.

Parameterized Mailboxes

- By default mailboxes are typeless. They can send and receive different data types. This may lead to runtime errors.

- Mailbox type can be parameterized by passing type as a argument.

```
mailbox #(string) mbox;
```

- In parameterized mailboxes, tools catches type mismatch errors at compilation time.

Example

```
module test;
mailbox mb;                          //typeless Mailbox

string s; int i;
initial begin
mb=new();                            //Unbound Mailbox
$monitor("s=%s and i=%0d at time=%0d", s, i, $time);
fork gen_data;
rec_data;
join end
endmodule
```

Example

```
task gen_data;
mb.put("Hello");
#3 mb.put(7);
#4 mb.put("Test");
#3 mb.put(3);
#3 mb.put("Hi");
#2 mb.put(9);
endtask
```

```
task rec_data;
#1  mb.peek(s);
#2  mb.get(s);
#2  mb.get(i);
#1  mb.peek(s);
#2 void'(mb.try_get(s));
#1 void'(mb.try_get(i));
#4 mb.get(s);
#2 mb.get(i);
endtask
```

Example

Result:

s= and i=0 at time=0

s=Hello and i=0 at time=1

s=Hello and i=7 at time=5

s=Test and i=7 at time=7

s=Test and i=3 at time=16

Example

```
module test;
mailbox #(int) mb;     //Parameterized Mailbox

int i;
initial begin
mb=new(3);             //bound mailbox
$monitor("i=%0d at %0d", i ,$time);
fork gen_data;
rec_data;
join end
endmodule
```

Example

```
task gen_data;
mb.put(1);
#1 mb.put(7);
#1 mb.put(4);
#2 mb.put(3);
#2 void’(mb.try_put(2));
#10 mb.put(5);
#2 mb.put(6);
endtask
```

```
task rec_data;
#1 mb.peek(i);
#5 mb.get(i);
#2 mb.get(i);
#2 void’(mb.try_get(i));
#1 mb.get(i);
#2 void’(mb.try_get(i));
#2 void’(mb.try_peek(i));
#2 mb.get(i);
endtask
```

Example

Result:

i=0 at time=0

i=1 at time=1

i=7 at time=8

i=4 at time=10

i=3 at time=11

i=2 at time=13

i=5 at time=18

- Create a class sequence_item contains following properties:
- Uart_data – 8 bit
- Gps_addr – 4bit
- Pkt_len – 3bit
- Create another class named as sequencer and with the help of concept of nested class and mailbox try send 10 randomize packet via this class
- Create another class driver which will receive the transmit packets from sequencer class via mailbox.

System Verilog

PROGRAM BLOCK & INTERFACE

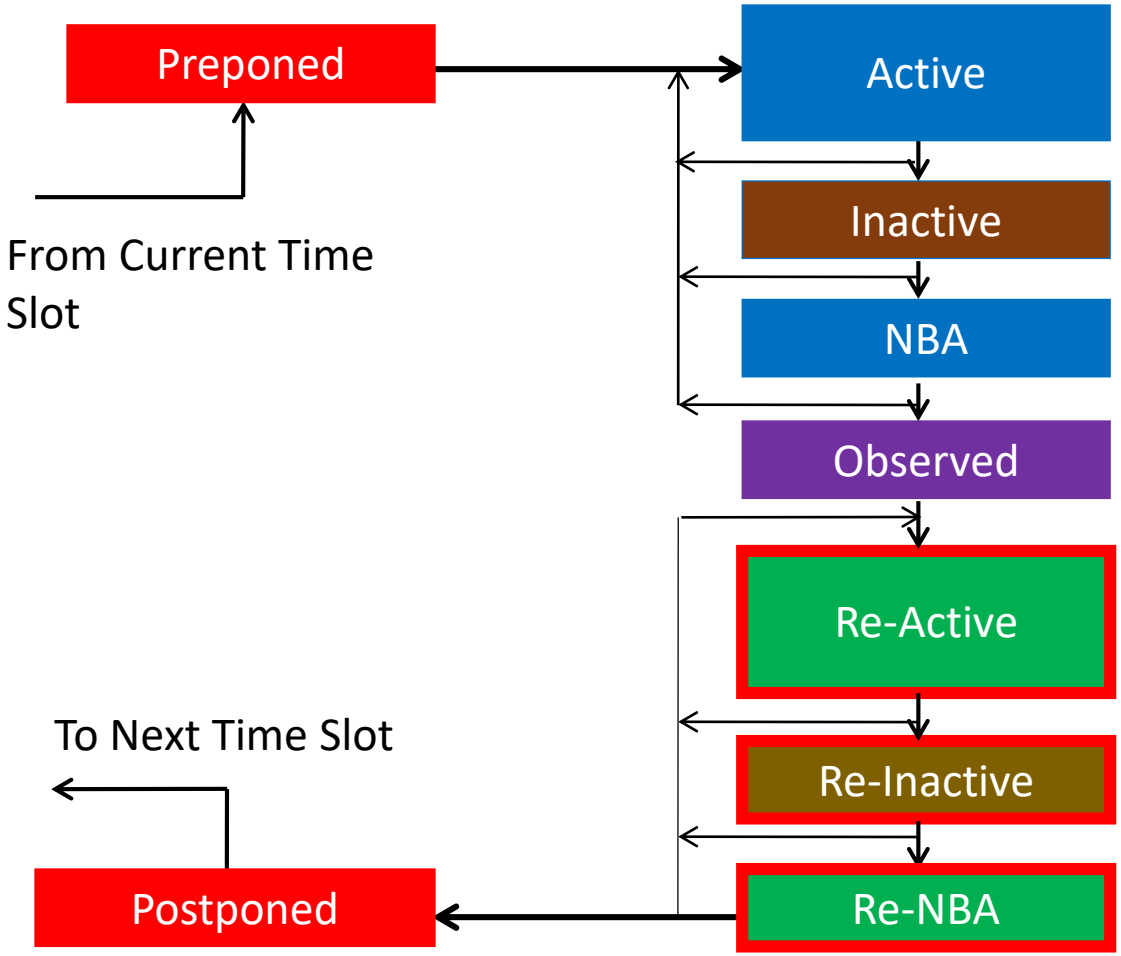
Program Block

- Verilog module works well for design but when used for Test benches may lead to race-around condition between design and Test bench.
- System Verilog adds program block which is used meant for writing Test Bench.
- program and endprogram keywords are used to define a program block.

Program Block

- program block has following features:
  - They separate test benches from design unit.
  - Statements are executed in Reactive Region.
  - always blocks are not allowed in program block.
  - They can be instantiated inside other modules.
  - Instance of module or program block is not allowed inside program block.
  - They have access to variables present inside a module where they are instantiated but vice versa is not true.
  - Implicit system task $finish is called when program block terminates.

Program Block Region



Test Bench runs once all design related activities are over.

Program Block Runs Here

Example1

```
module tff (q, clk, t);
input clk, t;
output reg q=0;

always @ (posedge clk)
if(t) q<= ~ q;
endmodule
```

```
module tb;
reg clk=0, t=1;
wire q=0;

always #5 clk=~clk;

tff u0 (q, clk, t);

always @ (posedge clk)
$display($time, “q=%d”, q);

endmodule
```

Example1

Result:

5 q= 0

15 q= 1

25 q= 0

35 q= 1

45 q= 0

55 q= 1

Example2

```
module tff (q, clk, t);
input clk, t;
output reg q=0;

always @ (posedge clk)
if(t) q<= ~ q;
endmodule
```

```
program tb (input clk);

initial       //always not allowed
begin
forever @ (posedge clk)
$display($time, “q=%d”, q);
end

initial t=1;

endprogram
```

Example2

```
module top;
reg clk=0, t;
wire q;

always #5 clk=~clk;

tff u0 (q, clk, t);
tb u1 (clk);                    //program has access to t and q

endmodule
```

Example2

Result:

5 q= 1

15 q= 0

25 q= 1

35 q= 0

45 q= 1

55 q= 0

Example3

```
program tb;
int a;
initial $monitor("result is %d", a);

initial begin
#3 a= a + 2;
#4 a= a + 3;
end

endprogram
```

Result : result is 2

$monitor does not execute for a=5 because of implicit $finish

Example4

```
program tb;
int a;
initial $monitor("result is %d", a);

initial begin
#3 a= a + 2;
#4 a= a + 3;
#1 ;
end

endprogram
```

Result :

result is 2

result is 5

- Write a code for parallel in serial out shift register for 8 bits
- Create a test-bench using program block