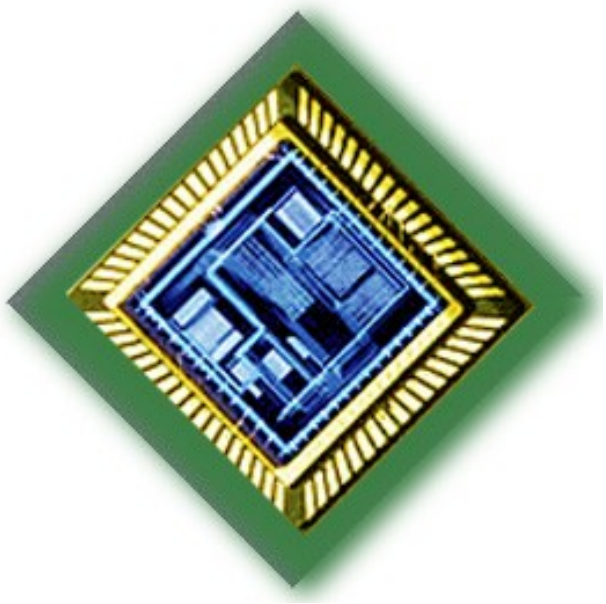


SystemVerilog para Verificação funcional com OVM

Curso do Brazil-IP

Elmar Melcher

UFCG

elmar@dsc.ufcg.edu.br

Roteiro

1)Introdução

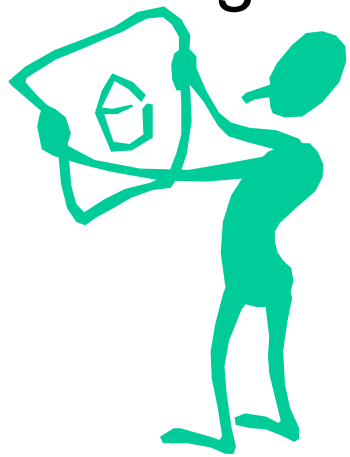
2)Fluxo de projeto

3)Representação gráfica vs. textual

4)Requisitos de uma linguagem de descrição de hardware

5)Um exemplo de uma descrição em SystemVerilog

6)Simulação com SystemVerilog



Roteiro

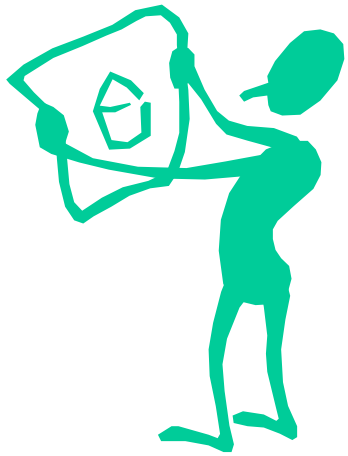
7) Descrição TLM em SystemVerilog
    1) Class
    2) FIFO
8) Testbench
    1) Transações
    2) Randomização
    3) Visualização
    4) Sinalização de erros

Roteiro

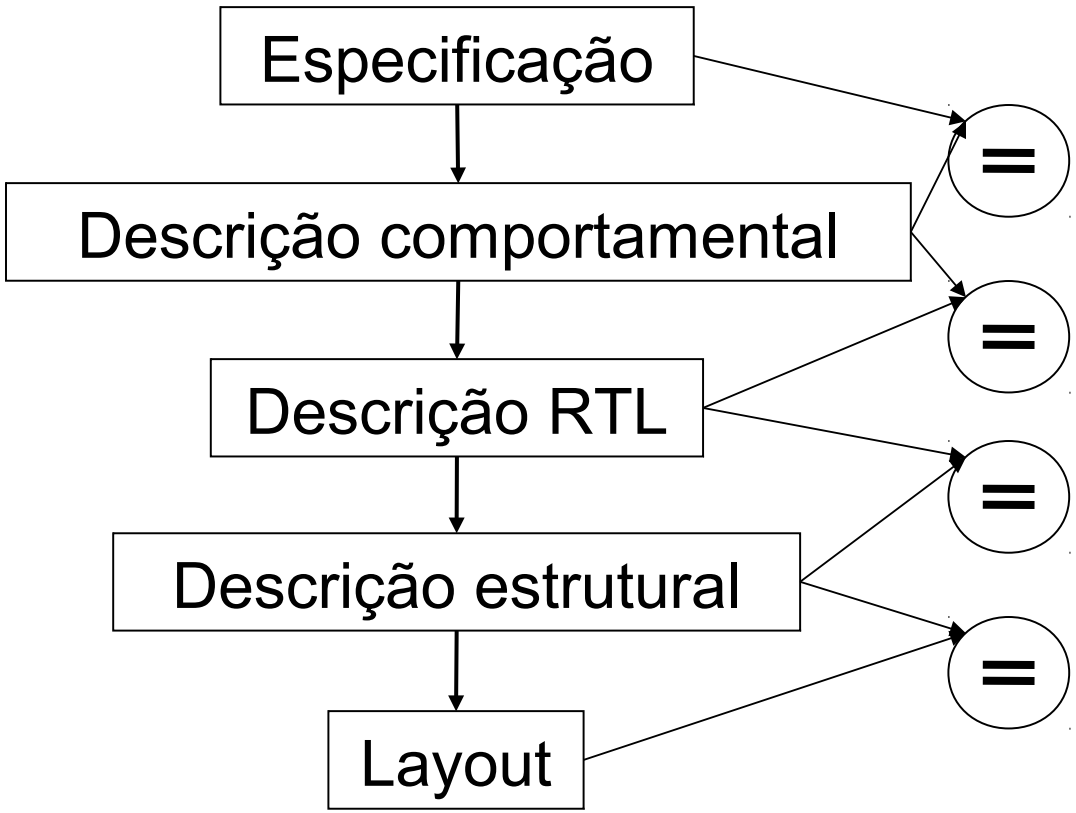
9) Cobertura de código
10) Cobertura funcional
11) Cosimulação com C

Introdução

1) Fluxo de projeto
2) Representação gráfica vs. textual
3) Requisitos de uma linguagem de descrição de hardware
4) Simulação com SystemVerilog



Fluxo de projeto (simplificado)



Função

Função & Timing

Função & Timing

Função & Timing

consumo, área, etc.

Propriedades desejáveis de uma HDL

- Expressar ações concorrentes
- Expressar tempo (atraso, clock)
- Permitir descrição comportamental, estrutural e física
- Permitir mesclar diferentes vistas de diferentes subsistemas
- Permitir simulação, síntese e verificação
- Ser fácil e seguro de usar

Exemplos de outras HDLs

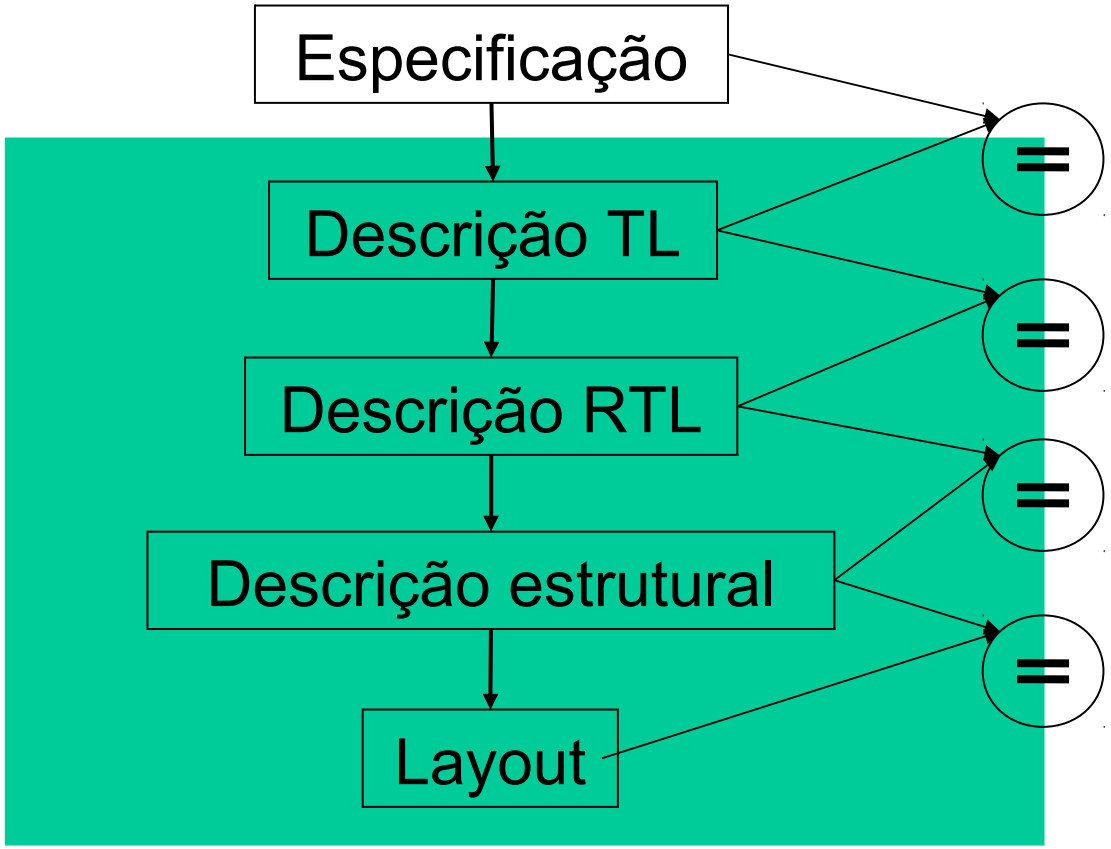
- Verilog 1995, 2001, 2005
- SystemC
- VHDL (VLSI HDL - Very Large Scale Integration Hardware Description Language)
- Abel, Palasm, Cupl, OCCAM, Handle-C, ELLA

. . .

Propriedades de SystemVerilog

- Expressar ações concorrentes
- Expressar tempo (atraso, clock)
- Permitir descrição TL ✓, RTL ✓, estrutural ✓ e física ✓
- Permitir mesclar diferentes vistas de diferentes subsistemas
- Permitir simulação ✓, síntese ✓ e verificação ✓
- Ser fácil e seguro de usar

Fluxo de projeto usando SystemVerilog



Função

Função & Timing

Função & Timing

Função & Timing

consumo, área, etc.

Breve Histórico

- Verilog, 1981
- SystemVerilog, padrão IEEE em 2005
- OVM, Janeiro 2008

Código bonito

- 50% é comentário
- Todas as declarações tem comentário
- Um **module** ou **class** não ultrapassa 100 linhas (uma página na tela)
- Identificadores refletem o que é o objeto identificado
- Indentação consistente

- regras do SRS da Motorola
- regras do Brazil-IP Network
  http://lad.dsc.ufcg.edu.br/fenix/metodologia

Código bonito

Identificadores

- Evite nomes longos demais:

  fpop_rs1 no lugar de **floating_point_opcode_rs1**

- Evite confusão entre '**0**' e '**O**', '**1**' e '**l**'.

- Use capitalização consistente.
  Estilo C: **packet_addr, data_in**
  Estilo Pascal: **PacketAddr, DataIn**
  Estilo Modula: **packetAddr, dataIn**

Código bonito

Sufixo de identificadores

***_clk**	sinal de relógio
***_next**	sinal antes de registrá-lo
***_n**	sinal ativo nível baixo
***_xi**	entrada do circuito
***_xo**	saída do circuito

Comentários

```
/* Comentario atravessando
   varias linhas */

// Comentario ateh o fim da linha
```

Cuidado com acentos á é ô à etc.

Convenções Léxicas

- **Números**

  Decimal, hexadecimal, binário com tamanho em bits:
  6'd33, 8'hA6, 4'b1101
  default: decimal positivo sem tamanho
  permitido usar _ + -
  8'b1001_0011

- **Cadeias de caracteres**

  "Delimite usando aspas numa mesma linha"
  Limitado a 1024 caracteres

Convenções Léxicas

- **Identificador**
  - A ... Z
  - a ... z
  - Underscore
- Primeiro caractere de um identificador não pode ser um dígito
- **SystemVerilog diferencia letras maiúsculas de minúsculas (case sensitive)**

Tipos de Dados

```
bit b;          //1 bit, só '1' e '0'
byte B;         //8 bits, só '1' e '0'
shortint;       //16 bits, só '1' e '0'
int i;          //32 bits, só '1' e '0'
longint l;      //64 bits, só '1' e '0'
shortreal r;    //32 bits, ponto flutuante
real r;         //64 bits, ponto flutuante
string s;       //string de caracteres
name name_i;    //instancia de classe
```

Tipos de Dados

Tipos sintetizáveis: default é sem sinal.

Tipos não sintetizáveis: default é com sinal.

Ex.: **logic [15:0] value;**
value = [0 : 65,536].

Ex.: **shortint value;**
value = [-32,768 : 32,767]

Operadores lógicos unários

~	negação bit a bit
!	negação lógica
&	“e” bit a bit
|	“ou” bit a bit
^	“ou” exclusivo
-	negativo
+	positivo

Operadores aritméticos binários

*	multiplicação
/	divisão
%	resto da divisão
+	soma
-	subtração

Operadores lógicos binários

- Deslocamento lógico (com preenchimento com ‘0’)
- Deslocamento negativo não pode

<< (esquerda) **>>** (direita)

- Deslocamento aritmético

<<< (esquerda) **>>>** (direita)

- Comparação

== **!=** **<** **<=** **>** **>=**

- Lógico

| **&** **^** **||** **&&**

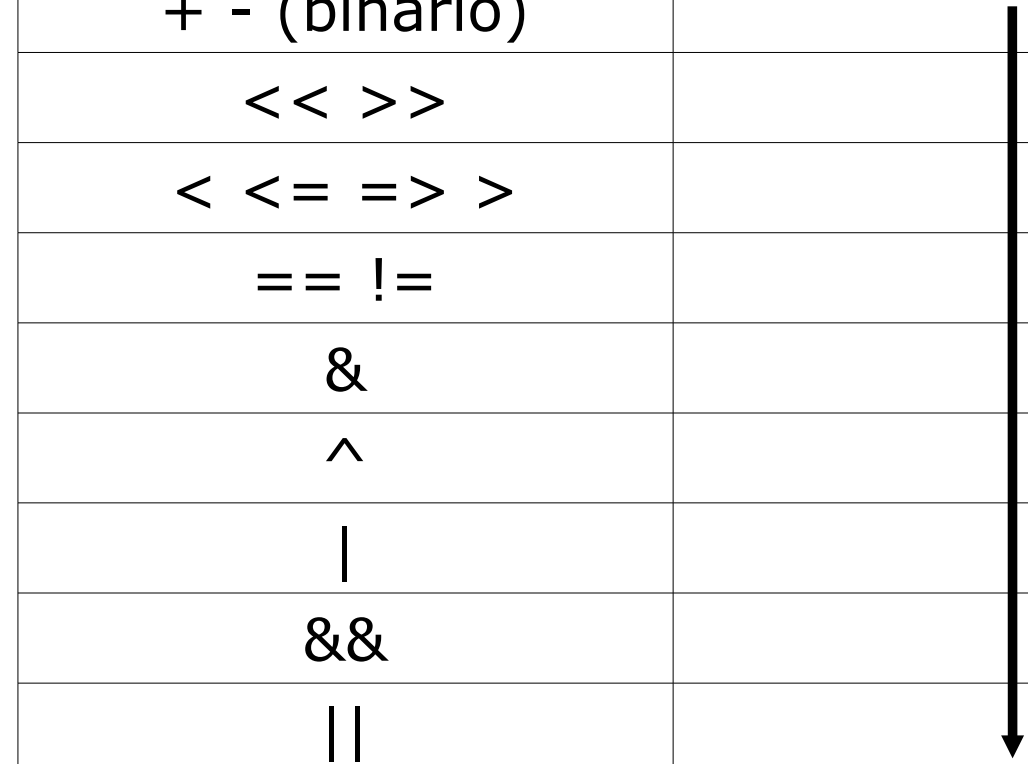
Operador condicional

<condição> ? <expressão_verdadeira> : <expressão_falsa>

- Para expressões simples é útil.
- Usá-lo aninhando é suicídio.
- Prefira estruturas **if ... else**.

Precedência de operadores

+ - ! ~ (unario)	mais
+ - (binario)	
<< >>	
< <= => >	
== !=	
&	
^	
|	
&&	
||	
?: (condicional)	menos



Diretivas de Compilação

- `define – (Similar a #define em C) usado para definir parâmetro global
- Exemplo:

```
`define HALF_PERIOD 500
always #(`HALF_PERIOD) clk <= !clk;
```

- `include – usado para incluir outro arquivo
- Exemplo:

```
`include "./refmod.svh"
```

Diretivas de Compilação

- parameter – usado para definir parâmetros globais; melhor que `define
- Exemplo:

```
parameter HALF_PERIOD = 500;
always #(HALF_PERIOD) clk <= !clk;
```

Simulação
a definição

Modelo:
Descrição de aspetos *interessantes* de um sistema real

Simulação:
Dado um modelo de condições de contorno (sinais de entrada), fornece a reação do sistema (sinais de saída) com respeito aos *aspetos modelados*

Simulação
como funciona

- Trata *eventos*.
  - Um evento é uma mudança de estado associado a um tempo simulado.
  - Um evento ocorre devido a outro evento.
- Vários eventos podem ocorrer ao mesmo tempo.
- Cuidado com a causalidade!

Simulação

o quê ela faz

- Processamento concorrente,
  faz de conta que diferentes ações acontecem no mesmo tempo.
- Obedece a especificações temporais,
  como atrasos.

"tempo de simulação"
vs.
"tempo simulado"

Simulação

- **Tempo de simulação**:

Tempo que a simulação dura ou demora para terminar. Tempo real.

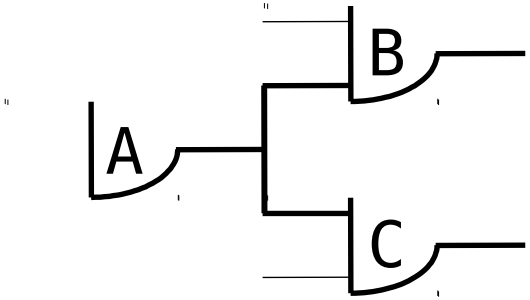
- **Tempo simulado**:

A simulação simula o hardware e o tempo de execução dele. Tempo virtual.

Simulação

um exemplo

Saída de A muda; isso leva B e C a serem executadas.



Muito embora, A não chama B nem C como numa chamada de função,

B e C executam porque são conectadas a saída de A

Atrasos na simulação

- **Sintaxe:**

  #(valor)(unidade de tempo)

- **Exemplo:**

```
task equal();
  if (a = b)
    $display("OK!");
  else
    $display("ERROR!");
endtask
initial #20ms equal();
```

Construções Procedurais

- Existem dois:
  - initial : Executa uma única vez no início da simulação, NÃO sintetizável
  - always_comb : Executa repetidamente, sintetizável

- Exemplo:

```
...
initial begin
  Sum <= 0;
  Carry <= 0;
end
...
```

```
...
always_comb begin
  Sum <= A ^ B;
  Carry <= A & B;
end
...
```

Fornecer sinais de entrada

```
program tb;

  logic clk;
  logic a,b,s,c;

  initial clk <= 0;
  always #5 clk <=
    ~clk;

  initial begin
    a <= 0; b <= 1;
    @(posedge clk);
    a <= 1; b <= 0;
  end

  duv U1(.*);

endprogram
```

Processamento concorrente

```
class transmitter;
   (...)
   task run();
      (...)
   endtask

endclass

class receptor;
   (...)
   task run();
      (...)
   endtask

endclass
```

```
program tb;

   (...)

   initial run_test();

   (...)

endprogram
```

Encerrando a simulação

```
program tb;

   (...)

   initial #1ms global_stop_request;

endprogram
```

Transação em SystemVerilog

Uma **ovm_transaction** class

Exemplo:

```
class packet extends ovm_transaction;
   enum {RD, WR} cmd;
   int addr;
   int data;
endclass
```

Referências a uma class são ponteiros.

- Um ponteiro é nulo antes de ser atribuído.
- O carroceiro do lixo vai passando...

FIFO

no módulo transmissor:

- declaração:

```
ovm_put_port #(packet) to_receptor;
```

- uso:

```
packet x;
x = new();
to_receptor.put(x);
```

no módulo receptor:

- declaração:

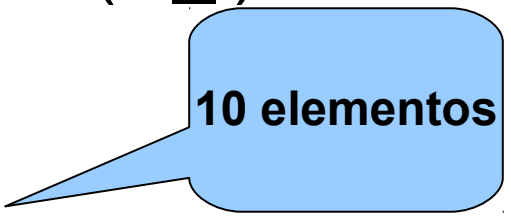
```
ovm_get_port #(packet) from_transmitter;
```

- uso:

```
packet b;
from_transmitter.get(b);
```

FIFO (cont.)

No módulo que instancia transmissor (tx_i) e receptor (rx_i):



- declaração:
  tlm_fifo #(packet) myfifo;
  myfifo = new("myfifo", this, 10);
- conexão:
  tx_i.to_receptor.connect(myfifo.put_export);
  rx_i.from_transmitter.connect(myfifo.get_export);

Transações e FIFOs

- Somente FIFOs (e não sinais) conectam source, driver, checker, modref e monitor.
- Transações são transportadas explicitamente de um módulo para outro, mas o *handshake* para garantir sincronismo é implícito.
- Metodologia simples, eficiente e aplicável para todas as situações.

FIFO Exemplo

- **source.sv:**

```
class source extends ovm_threaded_component;
  ovm_put_port #(packet) to_sink;
  function new(string name, ovm_component parent);
    super.new(name, parent);
    to_sink = new("to_sink", this);
  endfunction

  task run();
    packet p;
    forever begin
      p = new();
      assert(p.randomize());
      to_sink.put(p);
    end
  endtask
endclass
```

FIFO Exemplo

- **sink.sv:**

```
class sink extends ovm_component;
  ovm_get_port #(packet) from_source;
  function new(string name, ovm_component parent);
    super.new(name, parent);
    from_source = new("from_source", this);
  endfunction

  task run();
    packet pd;
    forever begin
      from_source.get(pd);
    end
  endtask
endclass
```

FIFO Exemplo

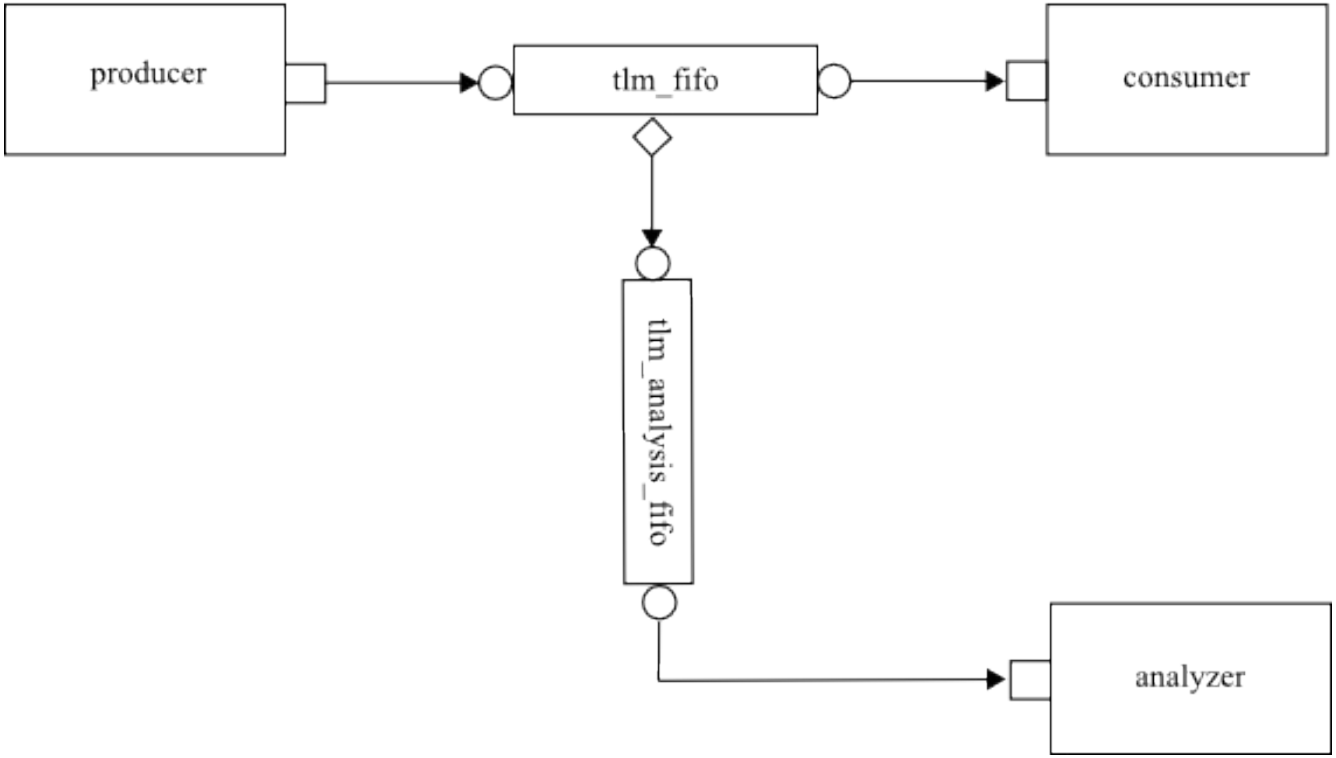
- **conector.sv:**

```
module conector;
  `include "trans.sv"
  `include "source.sv"
  `include "sink.sv"

  source source_i = new("source_i", null);
  sink sink_i = new("sink_i", null);

  tlm_fifo #(packet) source_sink = new("source_sink", null, 3);
  [...]
  initial begin
    source_i.to_sink.connect(source_sink.put_export);
    sink_i.from_source.connect(source_sink.get_export);
  end
endmodule
```

TWO HEAD FIFO



TWO HEAD FIFO

No módulo que instancia transmissor e receptor:

- declaração:

```
tlm_fifo #(acesso) fifo1;
fifo1 = new("fifo1", this, 10);

tlm_analysis_fifo #(acesso) fifo2;
fifo2 = new("fifo2", this);
```

TWO HEAD FIFO

No módulo que instancia transmissor e receptor:

- conexão:

```
src.out_port.connect(fifo1.put_export);
rm.in_port.connect(fifo1.get_export);

fifo1.put_ap.connect(fifo2.analysis_export);
driver.in_port.connect(fifo2.get_export);
```

Métodos para FIFO de OVM

Bloqueantes:

put, get, peek

Não bloqueantes:

try_put, try_get, try_peek

Não bloqueante requer o uso de delays. Ex.:

```
forever begin    //same as while(1)
   from_source.try_get(pd);
   #10ns;
end
```

Declarando elementos de um *testbench*

Exemplo: módulo transmissor

- Declaração:

```
class transmitter extends ovm_component;
  ovm_put_port #(packet) to_receptor;
  (...)
   function new(string name, ovm_component parent);
      super.new(name, parent);
      to_receptor = new("to_receptor", this);
   endfunction
   (...)
endclass
```

Visualização de formas de onda

Usando métodos de OVM

No início da task run():

recording_detail = OVM_FULL;

Antes de cada transação:

begin_tr (transaction, “label”, “descriptor”);

Após cada transação:

end_tr(transaction);

Visualização de formas de onda

Usando métodos de OVM

É preciso também configurar a gravação da transação, dentro da declaração da classe, redefinindo o método **do_record()**:

```
rand int value1;
rand int value2;
(...)
function void do_record (ovm_recorder recorder);

   recorder.record_field("name1", value1, $bits(value1), base);
   recorder.record_field("name2", value2, $bits(value2), base);
   (...)
endfunction
```

Sinalizar Erros

Usar

ovm_report_error(“nome_transacao”, psprintf(“esperado: %d recebido %d”, valor_refmod, valor_monitor))

que fornece o nome do módulo e o tempo simulado.

Incrementa um contador de erros, pode visualizar o contador junto às formas de onda e às transações.

Sinalizar Erros

Usar

record_error_tr(“Checker”);

Para visualizar nas formas de onda o tempo simulado em que o erro ocorreu.

Randomização direcionada de transações

```
class packet extends ovm_transaction;
  rand int i;
  constraint i_range {
    i > 0;
    i < 16;
  }
endclass
```

Randomização direcionada de transações

```
class packet extends ovm_transaction;
   rand int i;
   constraint i_range {
     i dist {[1:9] :/ 5, [10:15] := 4};
   }
endclass
```

Randomização direcionada de transações

```
packet sample; //conforme slide anterior
(...)
task run();
   (...)
   sample = new();
   assert(sample.randomize());
   (...)
endtask
```

Task *e* Function

Tasks têm domínio de tempo.

Functions são *timeless*.

Exemplos:

task my_task (int a, real b);

function longint my_function (int a, real b);

Concorrência (fork)

Possibilidades:

fork... join

fork... join_any

fork... join_none

disable_fork

wait_fork

Concorrência (fork)

Exemplo:

```
fork
   task_A();
   task_B();
join_any
fork
   task_C();
   task_B();
join_none
wait_fork
```

Cobertura funcional

O Plano de Verificação define as situações que devem ser simuladas.

A cobertura funcional é uma medida de quantas situações do plano de teste já foram simuladas.

Por definição, cobertura de 100% significa que a verificação funcional está concluída.

Cobertura funcional (cont.)

Estímulos são criados na forma de transações randômicos (**rand**),

Respostas são gerados pelo monitor e pelo modelo de referência,

TDriver e TMonitor podem medir cobertura funcional de transações e do *handshake* de sinais.

O Refmod deve medir a cobertura de funções executadas.

Cobertura Funcional

Covergroups

Os engenheiros de verificação podem observar os valores das variáveis agrupando-as em um ou mais grupos de cobertura (*covergroups)*.

Um *covergroup* é um agrupamento de pontos de cobertura (*coverpoints* – um grupo de variáveis cujos valores serão simultaneamente amostrados e contados.

Cobertura Funcional

Definindo um *covergroup*

```
covergroup cg1;
  coverpoint b {
     bins b1 = {0, 1, 2, 3};
     bins b2 = { [5:8], 9 };
  }
endgroup
```

variável

Valores que devem ser observados

Cobertura Funcional

Instanciando um *covergroup*

```
cg1 cg1_inst = new();
```

Define a instância **`cg1_inst`** do *covergroup* **`cg1`**

O uso de parênteses é opcional

Cobertura Funcional

Instanciando um *covergroup*

Um covergroup pode ser instanciado dentro de um módulo, uma interface, um bloco de programa, ou de uma classe.

Não é suportado por SystemVerilog:

Atribuições nulas a uma instância de um *covergroup*

```
cg1 cg1_inst = null;
```

Atribuições diretas entre duas instâncias de *covergroups*

```
cg1_inst1 = cg_inst2;
```

Cobertura Funcional

Definindo um *coverpoint*

As variáveis cujos valores devem ser observados são definidas como *coverpoints* dentro de um ou mais *covergroups*.

Durante a simulação os valores das variáveis definidas como *coverpoints* são medidos e armazenados na base de dados da cobertura.

Expressões de mais de 32 *bits* não são suportadas!

Cobertura Funcional

Caixas (*bins*)

Um *coverpoint bin* é usado para definir os valores que devem ser medidos e armazenados durante uma simulação.

A definição do intervalo de um *bin* pode ser em qualquer base (binária, decimal, hexadecimal, ou octal).

Adicionalmente, o intervalo de um *bin* pode ter uma constante ou uma expressão.

Cobertura Funcional

Caixas (*bins*), exemplos:

Para definir o intervalo como sendo os valores de 0 a 5 e 10:

```
bins b1 = {[0:5], 10};
```

Para definir o intervalo como sendo os valores de 0 a 5 e 9 a 14:

```
bins b1 = {[0:5], [9:14]};
```

Para definir um bin para cada valor, 0, 2, e 7 (bin vetorial):

```
bins b_vect[] = {0,2,7};
```

Cobertura Funcional

Caixas (*bins*), exemplos:

Para definir o intervalo como sendo os valores em hexadecimal de 0 a F:

```
bins b1 = {[‘h0:‘hF]};
```

Para definir o intervalo como sendo os valores menor que ou igual a 5, e 10:

```
bins b1 = {[$:5], 10};
```

Para definir o intervalo como sendo os valores maior que ou igual a 7:

```
bins b1 = {[7:$]};
```

Para definir o intervalo como sendo os valores pares entre 0 e 15:

```
wildcard bins b1 = {4'b???0};
```

Cobertura Funcional

Ignorando valores nos resultados de cobertura

Valores associados a um coverpoint podem ser ignorados na medição da cobertura especificando-os como **ignore_bins**.

Os valores especificados com **ignore_bins** são excluídos da medição de valores de *coverpoints* e não resultam em um incremento na contagem da cobertura.

Cobertura Funcional

Ignorando valores nos resultados de cobertura

```
ignore_bins ignore_vals = {2, 3};
```

Palavra-chave para definir valores a serem ignorados

Os valores 2 e 3 serão ignorados na medição da cobertura

Cobertura Funcional

Especificando valores como sendo ilegais

Valores associados a um coverpoint podem ser definidos como ilegais na medição da cobertura especificando-os como **illegal_bins**.

Se um valor de um coverpoint for detectado e tiver sido especificado com **illegal_bins**, a simulação gera um erro.

Cobertura Funcional

Especificando valores como sendo ilegais

```
illegal_bins ill_vals = {1, 2};
```

Palavra-chave para definir valores que serão considerados ilegais

Os valores 1 e 2 são ilegais e gerarão um erro se detectados

Cobertura Funcional

Diferenças e semelhanças entre **illegal_bins** e **ignore_bins**

O cálculo da cobertura para **illegal_bins** é exatamento o mesmo para **ignore_bins**.

A diferença entre os dois é que no caso de detectados illegal_bins, a simulação reporta um erro.

Não são reportados erros no caso de **ignore_bins**.

Illegal_bins têm precedência sobre quaisquer outros *bins*, o que implica que eles resultam em um erro em tempo de execução, mesmo caso eles estejam incluídos em um outro *bin*.

Cobertura Funcional

Cobertura cruzada (*cross-coverage*)

Cossimulação

- Aqui: SystemVerilog + C

  (VHDL e SystemVerilog pode ser feito também)

- Para quê?
  - Usar C para o modelo de referência e SystemVerilog para TL e RTL.

- Sintaxe usando a DPI
  - Exemplos:

```
import “DPI-C” real function cos(real x);
import “DPI-C” void function my_function(int y);
import “DPI-C” task my_task();
```

Cossimulação

- Usando a função ou *task* importada no código:
  - Como uma função ou *task* normal de SystemVerilog.
  - Exemplo:

```
cos_a = cos(a);
my_function(b);
```