UNIVERSIDADE FEDERAL DE SÃO CARLOS
Centro de Ciências Exatas e Tecnologia
Departamento de Computação

Laboratório de Arquitetura e Organização de Computadores 1

Semana(s) 01 e 02

Implementação de um *testbench* em Verilog [1] [2] [3] [4]

Aviso ENPE

Esta prática está adaptada para ser realizada em simuladores. Quando não for possível obter a informação solicitada devido à limitação do simulador adotado, o impedimento deve ser destacado no relatório.

1 Instruções Gerais

- Ler atentamente o procedimento antes de sua execução.;
- Atentar para prazos e formato de entrega dos resoltados.

2 Objetivos

Do roteiro:

- Descrever as atividades gerais desenvolvidas na atividade;
- Apresentar as funcionalidades de um *testbench*;

Da Prática:

- Implementação de circuitos com portas lógicas básicas por meio da linguagem Verilog;
- Implementação de um *testbench*.

[1] Documento adaptado das Práticas de Laboratório dos professores: Ricardo Menotti, Maurício Figueiredo, Edilson Kato, Celso Furukawa e Luciano Neris.
[2] Revisão 04/09/2020: Prof. Artino Quintino.
[3] Revisão 10/02/2021: Prof. Maurício Figueiredo.
[4] Revisão 10/01/2022: Prof. Maurício Figueiredo.

3 Materiais e Equipamentos

- simulador online EDA Playground, disponível em http://www.edaplayground.com.

4 Fundamentos teóricos

4.1 Multiplexadores em Verilog

Em sistemas computacionais, muitas vezes é necessário escolher dados entre várias fontes possíveis. Suponha que existam duas fontes de dados, fornecidas como sinais de entrada x1 e x2. Os valores desses sinais mudam com o tempo, talvez em intervalos regulares, portanto sequências de 0s e 1s são aplicadas em cada uma das entradas x1 e x2. Deseja-se projetar um circuito que produza uma saída com o mesmo valor que x1 ou x2, dependendo do valor de um sinal de controle de seleção. Portanto, o circuito deve ter três entradas: x1, x2 e s. Suponha que a saída do circuito será igual ao valor da entrada x1 se s = 0, e será igual a x2 se s = 1, conforme a tabela abaixo, Figura 1.

s	$f(s, x_1, x_2)$
0	x_1
1	x_2

Figura 1: Tabela-verdade compacta de um Multiplexador 2x1, [1].

A Figura 2, abaixo, mostra a tabela-verdade de um multiplexador 2x1.

$s\ x_1\ x_2$	$f(s, x_1, x_2)$
0 0 0	0
0 0 1	0
0 1 0	1
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	0
1 1 1	1

Figura 2: Tabela-verdade de um Multiplexador 2x1, [1].

A partir desta tabela-verdade, podemos chegar ao seguinte circuito, Figura 3.

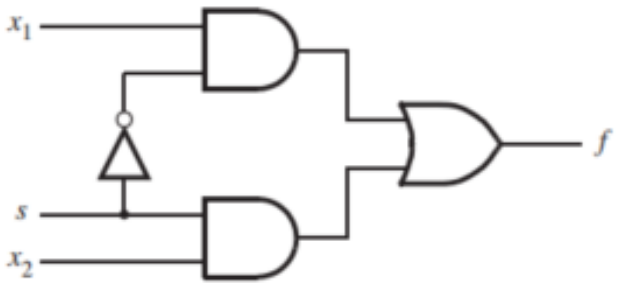

Figura 3: Multiplexador 2x1, [1].

Módulo

O circuito da Figura 3 pode ser descrito em linguagem Verilog em seu elemento básico, o módulo. Da mesma forma que os circuitos ; módulos possuem sinais de entrada e de saída.

A descrição do circuito fica como mostra o trecho de código abaixo, em que palavras reservadas da linguagem Verilog estão em verde (ou vermelho, no caso da declaração *wire*).

```
1 // Multiplexador 2x1
2
3 module exemplo1(x1, x2, s, f);
4   input x1, x2, s;              // por default, inputs e outputs são
```

```
5   output f;                    // wires de 1 bit
6
7   wire k, g, h;                // net interna (auxiliar)
8
9   not (k, s);
10  and (g, k, x1);
11  and (h, s, x2);
12  or (f, g, h);
13 endmodule
```

Após a palavra reservada **module** e o nome de identificação do módulo (exemplo1), vem a lista de sinais de entrada e saída. Por *default*, sinais tem dimensão de 1 bit. Comentários começam com // e se estendem até o final da linha.

Em Verilog, sinais são conhecidos como *nets* e a linguagem define vários tipos de *nets*. As que são usadas no módulo acima são do tipo *wire* : representam vias (ou trilhas, ou fios) que conectam entradas e saídas de circuitos. Por *default*, o Verilog assume que *nets* de "*input*" e "*output*" dos módulos são do tipo *wire*. OBS: Para evitar confusões, em *SystemVerilog* é introduzido o tipo *logic* cuja utilização faz com que o sintetizador fique responsável por esta especificação.

A declaração adicional das *"wires"* internas k, g e h (linha 7) não são realmente necessárias, mas o objetivo deste exemplo é descrever literalmente o diagrama lógico da Figura 3 em Verilog, além de melhorar a legibilidade do código e como exemplo de *nets* auxiliares. Neste primeiro exemplo, são usadas apenas as operações lógicas bit-a-bit (bitwise) NOT, AND e OR, representadas pelos comandos *not*, *and* e *and*.

Módulos devem terminar com a declaração **endmodule**.

Descrição por Atribuições *assign*

A palavra reservada *assign* serve para atribuir um sinal a uma *wire* e somente a *wire*. É a chamada *atribuição contínua*.Com ela pode-se definir como os sinais estão interligados em um circuito, por isso essa forma de representação é chamada também de descrição por fluxo de instrução (*dataflow*).

```
1 // Multiplexador 2x1
2
3 module exemplo2(input x1, x2, s,
4                 output f);
5
6     assign f = (x1 & ~s) | (x2 & s);
7
8 endmodule
```

Descrição Procedural com comandos *if-else* e *case*

Descrever cada conexão de circuito, sinal por sinal, no mínimo, é pouco produtiva. Em Verilog é possível descrever um circuito por meio de uma estratégia de mais alto nível - é a chamada *descrição procedural (behavioural)*, por meio de comandos (ou declarações) que servem para instruir o comnpilador Verilog como construir o circuito.

Um bloco procedural começa com *always* @, seguido de uma lista de sinais que servem para ativar o bloco, chamada de *lista de sensibilidade*. Os sinais devem ser separados pela palavra **or**. Também é possível utilizar a forma (*) para indicar "todos os sinais de entrada que aparecem no bloco" (alguns compiladores mais simples não aceitam essa sintaxe). Blocos de comando são delimitados por pares *begin – end*. Desde que o multiplexador do *exemplo 2* tem poucas entradas, é possível, de forma condicional, valorar as entradas para cada possibilidade e associá-las às saídas correspondentes. Da mesma forma que em linguagens de programação, isso é feito por comandos do tipo *if-else* e *case*. É o que mostra o exemplo a seguir.

```
1  // Multiplexador 2x1
2
3  module exemplo3(input x1, x2, s,
4                  output reg f);
5
6      always @(x1, x2, s)
7      if (s==0)
8          f = x1;
9      else
10         f = x2;
11
12 endmodule
```

DETALHE: Um *if* sem um bloco *else* ou um *case* que não explicita todas os casos possíveis. correspondem a condições em aberto as quais levam a circuitos divergentes com respeito à saída esperada. O "compilador"deve ser informado, via descrição do circuito, qual o sinal associado a TODAS as possibilidades de valores de entrada. Caso contrário O "compilador"assumirá uma associação presumida internamente diferente do esperado. Assim, a cada sentença *if* deve estar associada a uma sentença *else*. Da mesma forma, sentenças *case*, devem estar preenchidas ou conter a opção *default*.

4.2 Simulação com um *testbench*

Uma das etapas essenciais em projetos modernos de sistemas digitais é verificar a sua funcionalidade através de uma simulação. Para que um sistema digital seja simulado, deve haver um mecanismo que gere sinais conhecidos como *estímulos* para acionar o sistema, possibilitando observar o comportamento do circuito digital. Um estímulo é utilizado

como entradas do módulo que se deseja simular. O mecanismo para fazer isso em Verilog é chamado de *testbench* - em tradução direta para o português, bancada de teste.

Um *testbench* é apenas um módulo escrito em Verilog. Todavia, este módulo não é escrito da mesma forma com a qual se descreve um projeto de um circuito digital. Isso ocorre porque todo o projeto descrito em Verilog deve ser sintetizável, o que significa que ele terá um equivalente de hardware. Como o *testbench* será utilizado para simular o projeto, ele não precisa ser sintetizável.

Em termos de escrita, um *testbench* é um arquivo em Verilog que não possui entradas ou saídas e irá instanciar o projeto a ser testado como um módulo de nível inferior - lembre-se de que a linguagem Verilog segue um padrão hierárquico de codificação permitindo esse tipo de interação entre módulos. Esse módulo a ser simulado costuma ser chamado de *UUT (Unit Under Test)* - em alguns livros, *DUT (Device Under Test)*. O *testbench* gera os estímulos - ou seja, as condições de entrada - e as direciona para as portas de entrada do módulo UUT/DUT que está sendo testado. Verilog contém vários métodos para gerar padrões de estímulo. Como um *testbench* não será sintetizado, uma modelagem comportamental muito abstrata pode ser usada para gerar as entradas.

A saída do sistema pode ser impressa ou ser vista como uma forma de onda em uma ferramenta de simulação. Verilog também tem a capacidade de verificar as saídas em relação aos resultados esperados e notificar o usuário se ocorrerem diferenças.

A figura 4 abaixo ilustra pictoricamente a forma como o Testbench interage com o módulo a ser testado.

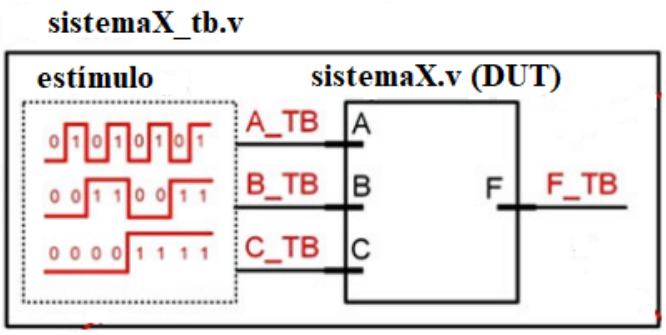

Figura 4: Diagrama de blocos para uma interpretação de um Testbench em Verilog, [1].

- É uma boa prática nomear um *testbench* com o mesmo nome do UUT/DUT, mas com "_tb"no final;
- Os estímulos são gerados em um *testbench* e acionam o UUT/DUT. Eles devem abranger todas as possibilidades de condições de entrada;
- O projeto a ser testado é instanciado no próprio *testbench*. Os sinais são declarados para se conectar como portas do UUT/DUT;

- As saídas podem ser impressas ou utilizadas para gerar gráficos.

Blocos do tipo *initial* (análogo a um bloco *always* sem lista de sensibilidade) e blocos *always* são adotados para descrever a geração de sinais (sinais elétricos que estimulam o circuito). Na codificação do Testbench o tempo é considerado explicitamente. Além disso, desde que o código do projeto corresponde a um circuito real, a simulação dos sinais deve ser tal que emule a sua natureza real, ou seja, acontece em paralelo.

```
1  // testbench para um multiplexador 2x1
2
3  `timescale 1ns/1ps        // unidade de tempo / precisão da simulação
4
5  module mux2x1_tb;
6
7    logic x1, x2;
8    logic s;
9    logic f;
10
11   exemplo1 UUT (x1, x2, s, f);        // Instanciação do módulo a simular
12
13   initial
14     begin
15       $dumpfile("dump.vcd");
16       $dumpvars(0);
17       $timeformat(-9, 0, "ns", 3);        // Tempo em ns (10E-9)
18
19       #10        // avança 10 unidades de tempo definida por timescale
20
21       x1 = 0;
22       x2 = 0;
23       s = 0;
24       $write( "%t: x1=%b, x2=%b, s=%b, f=%b\n", $time, x1, x2, s, f);
25
26       #10;
27
28       x1 = 1;
29       x2 = 0;
30       s = 0;
31       $write( "%t: x1=%b, x2=%b, s=%b, f=%b\n", $time, x1, x2, s, f);
32
33       #10;
34
35       x1 = 0;
36       x2 = 0;
37       s = 1;
```

```
38        $write( "%t: x1=%b, x2=%b, s=%b, f=%b\n", $time, x1, x2, s, f);
39
40        #10;
41
42        x1 = 0;
43        x2 = 1;
44        s = 1;
45        $write( "%t: x1=%b, x2=%b, s=%b, f=%b\n", $time, x1, x2, s, f);
46        #10;
47
48      end
49
50  endmodule
```

A simulação imprime os seguintes resultados na tela de console:

```
# KERNEL: 10ns: x1=0, x2=0, s=0, f=x
# KERNEL: 20ns: x1=1, x2=0, s=0, f=0
# KERNEL: 30ns: x1=0, x2=0, s=1, f=1
# KERNEL: 40ns: x1=0, x2=1, s=1, f=0
```

Algumas comentários sobre esse exemplo:

- A unidade de tempo e precisão temporal usada na simulação é definida na primeira linha pela diretiva. ‘timescale 1ns / 1ps // diretivas são comandos dados ao compilador e começam com ‘ (aspa simples).
- Para evitar as confusões a respeito de declarações de sinais e portas, SystemVerilog introduziu o *logic*.
- O comando #10 (linhas 20, 27, 34 e 41) faz o tempo de simulação avançar 10 unidades de tempo.
- No caso, fará a simulação avançar 10 ns, com precisão de 0,001 ns nos cálculos de propagação.

Neste exemplos há também comandos passados para o simulador (as chamadas *tasks* - tarefas de sistema).

- **$dumpfile()** O nome do arquivo onde salvar os sinais gerados (para posterior plotagem)
- **$dumpvars()** Os sinais a serem salvos. Se 0, salva todos os sinais.
- **$timeformat(-9, 0, “ns”, 3)** Define o formato como que o tempo de simulação é impresso.

  - No caso, em 10-9 s, com 0 dígitos depois da vírgula, “ns” como unidade e em 3 dígitos.

- Se não definido, o tempo é impresso na unidade definida como precisão em "timescale".

- **$time** Representa a variável interna do simulador que acumula o tempo de simulação.

- **$write()** Imprime uma mensagem na tela de console do simulador. O conteúdo "..." entre parêntesis segue as mesmas regras de formatação do comando "printf" da linguagem C.

- Além dos comandos citados acima, temos o comando **$finish** que encerra a simulação.

Os exemplos acima podem ser acessados pelo do simulador EDA Playground através do endereço https://www.edaplayground.com/x/5tHy

Simulação por varredura

Uma forma mais prática de se simular circuitos combinacionais é utilizar um contador que simule as possibilidades. Abaixo, Figura 5, temos um decodificador 1:2 com entrada de habilitação.

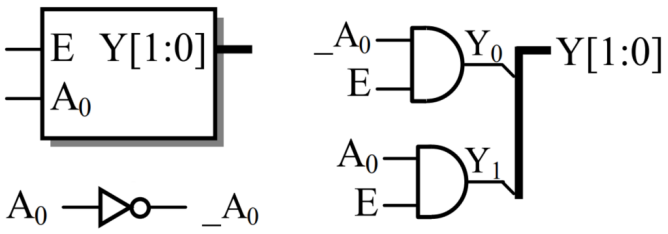

Figura 5: Símbolo e diagrama lógico do decodificador Dec12E, [1].

Utilizando a Descrição por Atribuições *assign*, este circuito em Verilog pode ser escrito da seguinte forma:

```
1  // Decodificador 1:2 com habilitação (enable)
2  // descrição por Atribuição assign
3
4  module Dec12E (
5    input E, A0,   // Por default, inputs e outputs são wires de 1 bit
6    output [1:0] Y // não tem virgula após o último parâmetro!
7          );       // mas tem ';' após fechar o parêntesis!
8
9    wire _A0, Y0, Y1;    // net interna (auxiliar)
10
```

```
  assign _A0 = ~A0;
  assign Y0 = E & _A0;
  assign Y1 = E & A0;
  assign Y[0] = Y0;  // NOTE ACIMA: ao declarar o bus Y,
  assign Y[1] = Y1;  // a dimensão [1:0] vem antes, mas na hora de usar
                     // um dos bit, o índice vem DEPOIS

endmodule // não tem ';' após endmodule!
```

Em seguida adota-se a estratégia de varredura para simular as n entradas (no caso, duas 2 entradas).

```
// Testbench do decodificador Dec12E_assign
// Simulação por varredura das entradas

`timescale 1ns / 1ps   // unidade de tempo / precisão da simulação

module Dec12E_tb( );

  reg A0, E;
  wire [1:0] Y;
  integer k;

  Dec12E UUT (E, A0, Y);  // Instanciação do módulo a simular

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(0);
    $timeformat(-9, 0, "ns", 3);  // Tempo em ns (10E-9)
    for (k = 0; k < 4; k = k+1 ) begin  // i gera todas combinações
                                        // das entradas
      A0 = k[0];
      E = k[1];
      #10;  // avança 10 unidades de tempo definida por timescale
      $write( "%t: E=%b, A0=%b => Y0=%b Y1=%b\n", $time, E, A0, Y[0],
             Y[1]);
    end; // for i
    $finish;
  end; // initial

endmodule
```

4.3 Simulação com EDA Playground

Os exemplos acima podem ser acessados pelo do simulador EDA Playground através do endereço https://www.edaplayground.com/x/5tHy e https://www.edaplayground.com/x/3Jpr.

Ele pode ser acessado com um navegador comum e é gratuito. É necessário se cadastrar e criar um login para poder salvar seus arquivos.

A princípio, o *testbench* pode ser digitado na janela *testbench.sv*, à esquerda, e o múdulo UUT/DUT em *design.sv*, à direita. O conteúdo desta última sempre é incluido na simulação. Mas é possível abrir outras janelas de código clicando no botão "+"e dar nomes como se fossem arquivos. Use a diretiva 'include "nome do arquivo"para flexibilizar essa regra.

Veja o tutorial em https://eda-playground.readthedocs.io/en/latest/tutorial.html ou o canal do YouTube em https://www.youtube.com/c/Edaplayground_EPWave/videos.

Atenção:

- Na coluna à esquerda da página web, selecione "Testbench + Design"e selecione *SystemVerilog/Verilog*.
- em "Tools", o compilador *default* (Aldec Riviera) e outros são para SystemVerilog.
- no EDA Playground, SALVE a página periodicamente e principalmente ANTES de sair dela ou fechá-la. A página não é salva automaticamente.
- na página "Profile" existem duas opções (muito) úteis:

  - o "Alert before leaving when code has been modified".

  - "Open EPWave waveforms on a separate page after run" (para não fechar a janela errada).

5 Atividades/Tarefas

5.1 Prática 01

PARTE A: Implementação de um testbench em um multiplexador 4x1
As tarefas são:

- Gerar o projeto em Verilog correspondente ao multiplexador 4x1;

- Testar todas as possibilidades, adotando a simulação por varredura;
- A saída da simulação corresponde a uma tabela com todos os casos simulados;

PARTE B: Projeto e Análise

Considere as variáveis "a"e "c"do tipo reg e, "b"e "d"do tipo logic (entradas do circuito), Considere duas sentenças inseridas no código espacialmente consecutivas em um bloco "always", voltado à descrição de um circuito (ver em seguida). Ou seja, o circuito corresponde à descrição feita pelas duas sentenças.

Considere ainda que a lista de sensibilidade seja definida por borda de subida, unicamente com os sinais de relógio e de reset.

No instante T = 1s, os seguintes valores das variáveis são verificados: "a = 1", "b = 2", "c = 1"e "d = 2". As entradas permanecem constantes a partir de então.

Considere dois casos:

- CASO 1: Sentenças organizadas na disposição em seguida:
  "c = b + d"
  "a = b + c"
- CASO 2: Sentenças organizadas na disposição em seguida:
  "c <= b + d"
  "a <= b + c"

As tarefas são:

- Apresentar a simulação de cada um dos circuitos. Apresentar uma saída no formato de tabela (valores decimais) para os sinais "a", "c", "b"e "d"imediatamente antes da primeira borda de subida de relógio e imediatamente após as bordas de descida do relógio. Além dessas variáveis, também apresentar os valores dos sinais na entrada dos registradores (não o relógio, não o reset) correspondentes, para os mesmos instantes. Também apresentar o tempo de relógio em todos os registros. Não é aceitável valores "x"ou "z";
- A primeira borda de subida, pare efeitos de resultados, acontece após a inicialização das variáveis;
- Considerar pelo menos cinco bordas de descida após as respectivas bordas de subida;
- A saída deve corresponder a uma tabela com os valores dos resultados nos momentos solicitados;
- Justificar o comportamento dos circuitos no relatório.
- Alterar a ordem das sentenças. Simular e analisar.

- Elaborar um relatório simplificado contendo: página de apresentação e duas seções correspondentes às partes A e B da atividade prática. Na primeira seção, apenas a URL deve ser apresentada. Na segunda seção, apresentar: a URL, o esquema dos circuitos, identificando a posição das variáveis que se apresentam na simulação e os comentários pertinentes após comparação de resultados, sob a perspectiva das similaridades das sentenças que os definem.

Referências Bibliográficas

[1] B. J. LaMeres, "Introduction to logic circuits & logic design with verilog," 2019.