



System Verilog

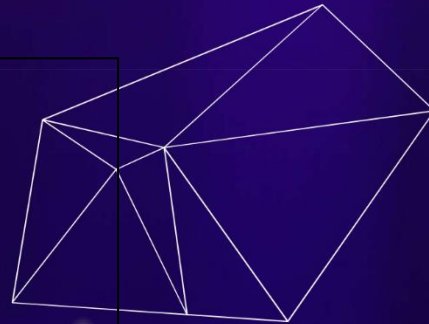
Unidade 4 | Capítulo 1

Professores

Antônio Gabriel Borralho
Danúbia Soares Pires
Diego Dutra Sampaio
Felipe Gomes Barbosa
Francisco Borges Carreiro
Matheus Silva Pestana
Orlando Donato Rocha Filho

Coordenação:

Cláudio Leão Torres
Jorge Renato dos Santos Silva
Waldenisson Novaes Carneiro
Washington Luis Santos Silva



Executores:



Coordenação:



Iniciativa:



Sumário

Unidade 4.....	3
Capítulo 1.....	3
1. BOAS-VINDAS	3
2. EDA Playground	4

Unidade 4

Capítulo 1

1. BOAS-VINDAS

Olá, estudantes!

Sejam bem-vindos ao nosso material de apoio sobre **SystemVerilog e suas aplicações práticas**. Este conteúdo foi elaborado para complementar o Capítulo 1 da Unidade 4, oferecendo explicações detalhadas, tutoriais e exemplos que facilitam a compreensão da linguagem e sua utilização em projetos digitais.

O objetivo deste material é apresentar um guia claro e acessível para o uso da plataforma **EDA Playground**, permitindo que vocês pratiquem simulações diretamente no navegador, sem a necessidade de instalação de ferramentas adicionais. Além disso, reunimos uma série de exemplos práticos — desde operações básicas com wire, reg e logic até aplicações mais avançadas com arrays, filas (queue), estruturas (struct) e máquinas de estados (enum) — que demonstram como o SystemVerilog pode ser usado para modelar sistemas digitais de maneira eficiente e organizada.

2. EDA Playground

O EDA Playground é uma plataforma gratuita e baseada em nuvem que permite escrever, simular e compartilhar códigos em linguagens de descrição de hardware como Verilog e SystemVerilog sem a necessidade de instalar softwares no computador. Seu uso é especialmente valioso no processo de aprendizagem, pois oferece um ambiente acessível e padronizado para todos os alunos, facilitando a prática de conceitos fundamentais de modelagem e simulação de circuitos digitais. Neste capítulo, exploraremos passo a passo como utilizar a plataforma, desde a criação de projetos simples até a visualização de formas de onda, de modo que cada estudante desenvolva autonomia para realizar experimentos e validar seus códigos de forma prática e interativa.

2.1. Primeiros Passos

As figuras a seguir guiarão no passo a passo para criar sua conta e rodar seu primeiro código exemplo, assim como mostram as funcionalidades das opções mais importantes. Ao acessar a primeira vez, veremos uma tela que nos auxiliará na criação da conta ou no login de uma conta já existente.

1º Passo: Criar conta – podemos criar uma conta genérica ou utilizar uma conta Google pré-existente.

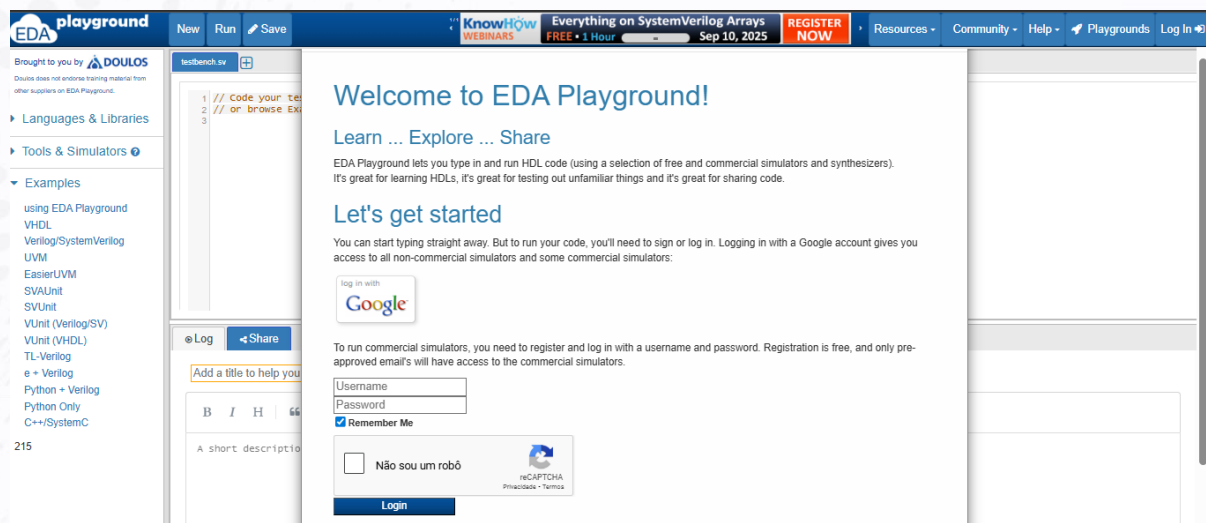


Figura 1 - Tela do Primeiro acesso ao EDA Playground

2º Passo: Ao acessar o **EDA Playground**, o estudante se depara com uma tela organizada em áreas principais que facilitam a criação e execução de códigos em Verilog ou SystemVerilog. No centro até a borda direita encontra-se o **editor de código**, onde é possível digitar ou colar programas, criar arquivos adicionais e organizar módulos (design.sv) e testbenches (testbench.sv). No lado esquerdo há o menu **"Tools & Simulators"**, utilizado para selecionar a linguagem desejada e o simulador compatível, como o *Icarus Verilog (SV)*, que será o mais utilizado. Logo acima do editor estão os **botões de ação**:

- **Run** (executar a simulação)
- **Save** (salvar e gerar um link para compartilhar o projeto)
- **New** (criar um novo arquivo)

A parte inferior da tela corresponde à **área de saída (Log)**, onde aparecem os resultados das simulações, mensagens de depuração ou erros de compilação.

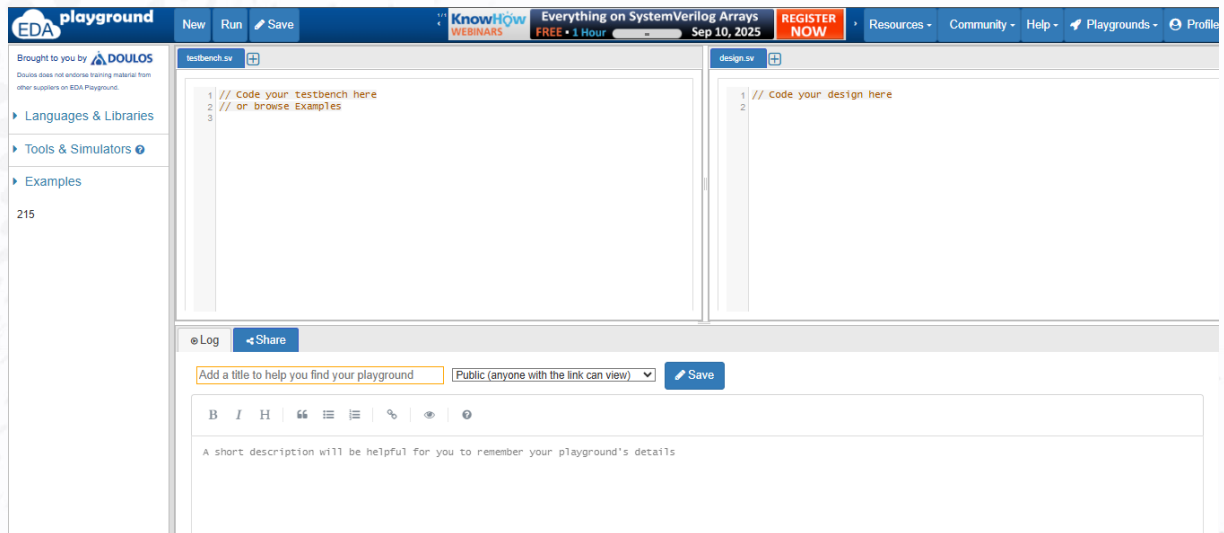


Figura 2 - Tela inicial do EDA Playground

3º Passo: No menu **"Languages & Libraries"**, selecionaremos a linguagem **SystemVerilog**, que será utilizada para desenvolver e simular os códigos durante as atividades. Já no menu **"Tools & Simulators"**, escolheremos principalmente o simulador **Icarus Verilog (SV)**, por ser gratuito, compatível com SystemVerilog e suficiente para executar os exemplos deste curso. Além disso, nesse mesmo menu é possível habilitar a opção **EPWave waveform viewer**, que permite visualizar graficamente as formas de onda geradas pela simulação.

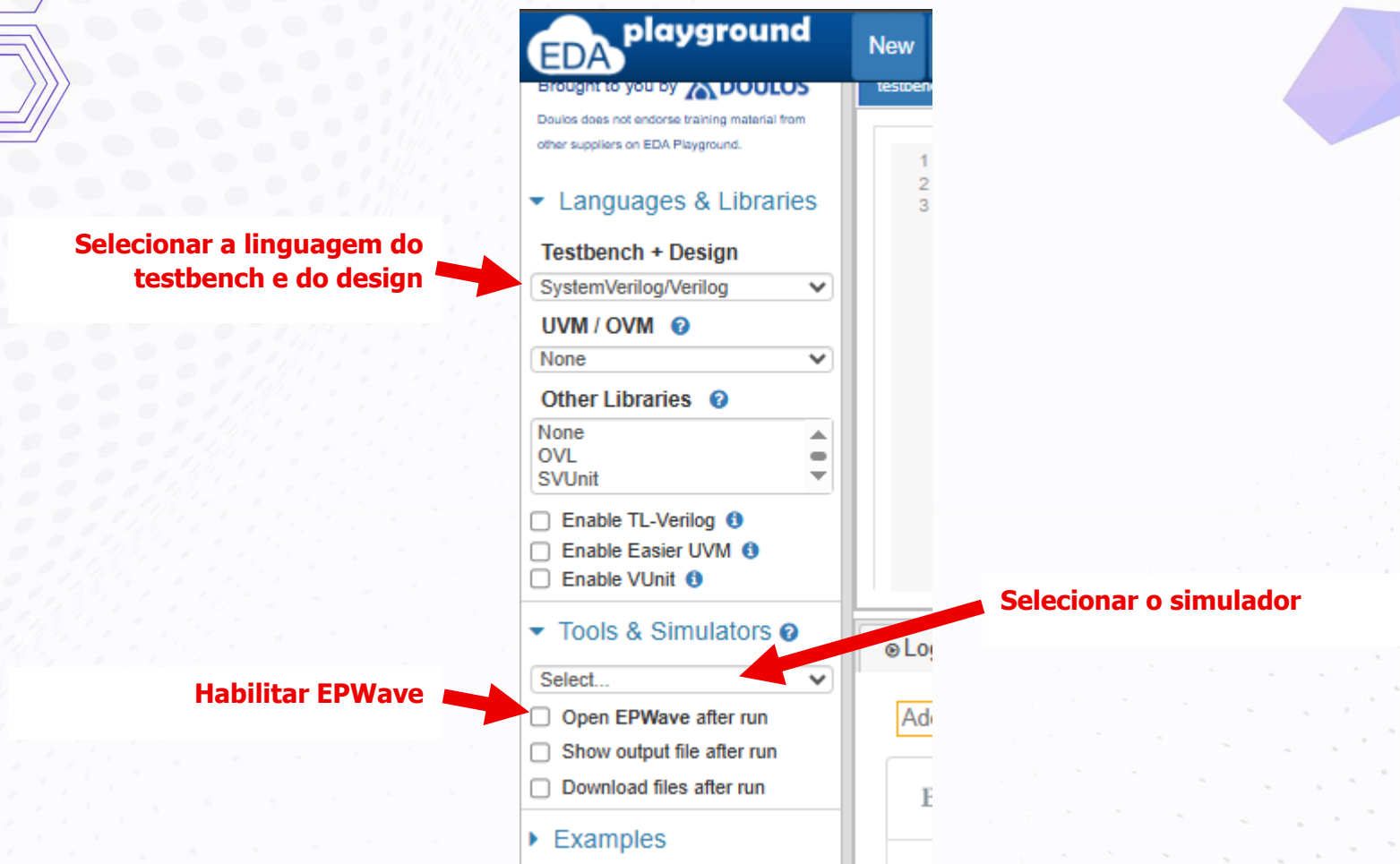


Figura 3 - Selecionando a linguagem, o simulador utilizado e habilitando o EPWave

4º Passo: Agora iremos rodar um código de teste abaixo. Selecione o simulador Icarus Verilog 12.0, coloque o código no campo design.sv, clique em Save, depois clique em Run.

```
module hello;  
  initial begin  
    $display("Hello, FPGA World!");  
    $finish; // encerra a simulação  
  end  
endmodule
```



Figura 4 - Hello World em System Verilog. Note que como não geramos um arquivo .vcd o EPWave não é inicializado.

2.2. Exemplo com EPWave: Meio Somador

Agora um exemplo de circuito simples: **Meio-Somador (Half Adder)**.

design.sv:

```
module half_adder(
    input  logic a, b,
    output logic sum, carry
);
    assign sum  = a ^ b; // soma = XOR
    assign carry = a & b; // carry = AND
endmodule
```


testbench.sv:

```
module tb_half_adder;  
    logic a, b, sum, carry;
```

```
// Instancia o módulo
```

```
half_adder uut (.a(a), .b(b), .sum(sum), .carry(carry));
```

```
initial begin
```

```
    $display("a b | sum carry");
```

```
    $display("-----");
```

```
    a = 0; b = 0; #10;
```

```
    $display("%b %b | %b  %b", a, b, sum, carry);
```

```
    a = 0; b = 1; #10;
```

```
    $display("%b %b | %b  %b", a, b, sum, carry);
```

```
    a = 1; b = 0; #10;
```

```
    $display("%b %b | %b  %b", a, b, sum, carry);
```

```
    a = 1; b = 1; #10;
```

```
    $display("%b %b | %b  %b", a, b, sum, carry);
```

```
    $finish;
```

```
end
```

```
endmodule
```

Saída esperada:

```
a b | sum carry
```

```
-----
```

```
0 0 | 0  0
```

```
0 1 | 1  0
```

```
1 0 | 1  0
```

```
1 1 | 0  1
```

Visualizando Ondas (GTKWave Online)

Adicione no **testbench** os comandos:

```
initial begin
```

```
    $dumpfile("waveform.vcd"); // gera arquivo de ondas
```

```
    $dumpvars(0, tb_half_adder);
```

```
end
```

Rode novamente e o EPWave abrirá automaticamente na mesma janela. Assim você verá os sinais a, b, sum, carry em forma de onda no tempo.

2.3. Comparando wire, reg e logic no SystemVerilog

No EDA Playground é possível organizar um projeto em vários arquivos, cada um representando um módulo diferente. Para isso, basta clicar em “+” na lateral direita da tela, criando novas abas de código além do design.sv.

Em cada aba pode ser escrito um módulo separado, como por exemplo *comb_with_wire.sv*, *mux_with_reg_legacy.sv* e *dff_with_logic.sv*. O módulo de integração, chamado top, deve ser colocado no arquivo principal design.sv, pois é ele que conecta todos os outros blocos em um sistema único.

Já o arquivo testbench.sv, à esquerda, continua sendo usado apenas para o código de teste. Dessa forma, o projeto fica organizado: cada componente em seu próprio arquivo, o top no design.sv e o ambiente de simulação no testbench.sv.

Crie os arquivos utilizando o botão, conforme a figura abaixo:



Figura 5 - Criando múltiplos arquivos no EDA Playground

1. comb_with_wire — exemplo com wire

```
module comb_with_wire(  
    input  logic a, b,  
    output logic y  
);  
    // NET (conexão física)  
    wire and_ab;  
  
    // Atribuições contínuas (característica de NETs)  
    assign and_ab = a & b;  
    assign y      = and_ab | b;  
endmodule
```

Neste módulo é declarado um sinal intermediário chamado `and_ab` do tipo `wire`. Esse tipo de dado é usado para representar conexões físicas, funcionando como um fio que transmite o resultado de uma operação lógica para outro ponto do circuito. O código mostra duas atribuições contínuas: primeiro, `and_ab` recebe o valor da operação `a & b`, que é um **AND lógico** entre as entradas.

Em seguida, a saída `y` é definida como `and_ab | b`, ou seja, o resultado do **OR lógico** entre `and_ab` e o sinal de entrada `b`. Essas

atribuições contínuas (assign) descrevem lógica combinacional pura, onde as saídas mudam instantaneamente conforme mudam as entradas.

O exemplo deixa claro que o wire não guarda estado: ele apenas transmite valores de forma imediata, simulando exatamente o comportamento de um fio real em hardware.

2. mux_with_reg_legacy — exemplo com reg

```
module mux_with_reg_legacy(  
    input logic a, b, sel,  
    output reg y  
);  
  
    // 'reg' é variável procedural (legado).  
  
    always @* begin  
        if (sel) y = a;  
        else y = b;  
    end  
endmodule
```

O módulo implementa um **multiplexador 2x1**, que escolhe entre as entradas a e b de acordo com o seletor sel. Aqui, a saída y

foi declarada como reg porque ela recebe valores dentro de um bloco procedural always. No bloco always @*, o @* significa que ele é sensível a qualquer mudança nas variáveis utilizadas dentro do processo, garantindo que y seja atualizado sempre que a, b ou sel mudarem.

Dentro do bloco, a instrução condicional if (sel) y = a; else y = b; faz a escolha: se sel vale 1, a saída será igual a a; caso contrário, será igual a b. A palavra-chave reg não significa necessariamente registrador físico neste caso, mas sim que a variável pode armazenar temporariamente o valor atribuído até a próxima avaliação do bloco procedural.

Isso é um ponto importante para compreender como o Verilog clássico lidava com atribuições em processos, e por que o SystemVerilog trouxe o tipo logic para unificar e simplificar essa semântica.

3. dff_with_logic — exemplo com logic

```
module dff_with_logic(  
    input logic clk, rst_n, d,  
    output logic q  
);  
  
    always @(posedge clk or negedge rst_n) begin  
        if (!rst_n) q <= 1'b0;
```

```
    else      q <= d;
  end
endmodule
```

Este código descreve um **flip-flop tipo D com reset assíncrono ativo em nível baixo**. As entradas são clk (clock), rst_n (reset ativo em 0) e d (dado de entrada), enquanto a saída é q. O bloco always @(posedge clk or negedge rst_n) indica que o processo será executado sempre que ocorrer uma borda de subida do clock ou uma borda de descida do reset.

Dentro do bloco, a primeira condição verifica se o reset está ativo (if (!rst_n)). Se estiver, q é forçado a 0 imediatamente. Caso contrário, na próxima borda de subida do clock, q assume o valor da entrada d. A atribuição <= é não bloqueante, o que é a prática correta em lógica sequencial.

O uso de logic no lugar de reg é a forma moderna de declarar variáveis em SystemVerilog, pois ele pode ser utilizado tanto em atribuições contínuas quanto procedurais, desde que respeitada a regra de apenas um driver por sinal. Este exemplo mostra como descrever corretamente um elemento de memória fundamental em circuitos digitais: o flip-flop.

4. design.sv — top — integração dos módulos

```

`include "comb_with_wire.sv"
`include "mux_reg_with_legacy.sv"
`include "dff_with_logic.sv"

module top(
    input  logic a, b, sel,
    input  logic clk, rst_n,
    output logic y_comb,
    output logic y_mux,
    output logic q_dff
);

    comb_with_wire    u_comb(.a(a), .b(b), .y(y_comb));

    mux_with_reg_legacy u_mux (.a(a), .b(b), .sel(sel), .y(y_mux));

    dff_with_logic    u_dff (.clk(clk), .rst_n(rst_n), .d(a ^ b),
    .q(q_dff));

endmodule

```

O módulo top reúne os três módulos anteriores em um único sistema. Ele recebe as entradas a, b, sel, clk e rst_n, e conecta cada componente instanciado às saídas correspondentes. O comb_with_wire calcula a saída combinacional y_comb usando

atribuições contínuas e um wire. O `mux_with_reg_legacy` gera a saída `y_mux` a partir de um processo procedural com `reg`. Já o `dff_with_logic` gera `q_dff`, que é a saída sequencial controlada pelo clock e pelo reset, recebendo como entrada $a \wedge b$, ou seja, o resultado de uma operação XOR entre `a` e `b`.

Esse exemplo ilustra a hierarquia de design no SystemVerilog: em vez de escrever todo o circuito em um único módulo, componentes menores e especializados são organizados e interligados em um nível superior. Essa abordagem torna os projetos mais claros, reutilizáveis e próximos da prática em projetos reais de hardware.

5. tb — testbench para validação

```
module tb;

    // Estímulos

    logic a, b, sel, clk, rst_n;

    // Observações

    logic y_comb, y_mux, q_dff;

    // DUT

    top dut(
        .a(a), .b(b), .sel(sel),
```

```
.clk(clk), .rst_n(rst_n),  
.y_comb(y_comb), .y_mux(y_mux), .q_dff(q_dff)  
);
```

```
// Clock 100 MHz (período 10 ns)
```

```
initial clk = 1'b0;
```

```
always #5 clk = ~clk;
```

```
// Dump para EPWave/GTKWave
```

```
initial begin
```

```
    $dumpfile("waveform.vcd");
```

```
    $dumpvars(0, tb);
```

```
end
```

```
// Estímulos e logs
```

```
initial begin
```

```
    rst_n = 0; a = 0; b = 0; sel = 0;
```

```
    #12 rst_n = 1;
```

```
    repeat (6) begin
```

```

#7 a  = $urandom_range(0,1);
    b  = $urandom_range(0,1);
    sel = $urandom_range(0,1);

#3    $display("t=%0t | a=%0b b=%0b sel=%0b ||
y_comb=%0b y_mux=%0b q_dff=%0b",

           $time, a, b, sel, y_comb, y_mux, q_dff);

end

#20 $finish;

end

endmodule

```

O testbench tb é responsável por simular o circuito descrito no módulo top. As variáveis a, b e sel são entradas de estímulo, enquanto y_comb, y_mux e q_dff são as saídas observadas. O clock é gerado por um processo que alterna o valor de clk a cada 5 unidades de tempo, resultando em um período de 10 unidades, equivalente a 100 MHz.

O reset é inicialmente colocado em 0 para forçar o circuito a um estado conhecido e, após 12 unidades de tempo, é liberado (rst_n = 1). Em seguida, o testbench aplica valores aleatórios a a, b e sel usando \$urandom_range, repetindo o processo várias vezes. Cada

conjunto de estímulos e respostas é exibido no console com `$display`, permitindo acompanhar o comportamento do circuito. Além disso, são gerados arquivos de forma de onda (waveform.vcd) que podem ser analisados graficamente no EPWave ou GTKWave.

Esse testbench mostra na prática como preparar um ambiente de verificação completo: geração de clock, aplicação de reset, estímulos variados, coleta de resultados e análise de ondas.

2.4. Banco de Registradores com Array Fixo

O código a seguir modela um banco de **8 registradores de 16 bits** com **escrita síncrona** e **duas portas de leitura combinacional**. O objetivo é mostrar, de forma prática, como **arrays fixos** representam memória de registradores, como **parâmetros** (WIDTH, DEPTH) generalizam o módulo, e como **\$clog2** dimensiona automaticamente o tamanho do endereço. O fluxo é hierárquico: o arquivo design.sv integra o sistema, “puxando” os módulos com diretivas ``include`; o register_bank.sv guarda a lógica do banco; e o tb.sv é o ambiente de verificação.

design.sv – top

```
`include "register_bank.sv"
```

```
module top #(
```

```
    parameter int WIDTH = 16,
```



```

parameter int DEPTH = 8
)(
    input logic          clk,
    input logic          rst_n,

    // Interface de escrita
    input logic          we,
    input logic [$clog2(DEPTH)-1:0] waddr,
    input logic [WIDTH-1:0] wdata,

    // Interfaces de leitura (duas portas)
    input logic [$clog2(DEPTH)-1:0] raddr_a,
    output logic [WIDTH-1:0] rdata_a,
    input logic [$clog2(DEPTH)-1:0] raddr_b,
    output logic [WIDTH-1:0] rdata_b
);

```

```

register_bank #(.WIDTH(WIDTH), .DEPTH(DEPTH)) u_rf (
    .clk    (clk),
    .rst_n  (rst_n),

```

```
.we    (we),  
.waddr (waddr),  
.wdata (wdata),  
.raddr_a (raddr_a),  
.rdata_a (rdata_a),  
.raddr_b (raddr_b),  
.rdata_b (rdata_b)  
);
```

```
endmodule
```

Este arquivo é o **ponto de integração** do design. Logo na primeira linha, a diretiva ``include "register_bank.sv"` torna o módulo `register_bank` visível para o simulador a partir do `design.sv`. Em seguida, o módulo `top` apenas **instancia** o banco de registradores e **expõe** sua interface (clock, reset, escrita e duas leituras).

Essa separação permite reutilizar `register_bank.sv` em outros projetos, enquanto o `top` permanece como camada de conexão de sinais e parametrização. Em ambientes como o EDA Playground, o simulador **compila a partir de `design.sv`**, então os include são necessários para que os outros módulos sejam encontrados.

register_bank.sv — banco de registradores com array fixo

```
module register_bank #(
    parameter int WIDTH = 16,          // Largura de cada registrador
                                         (bits)
    parameter int DEPTH = 8            // Quantidade de registradores
)(
    input logic          clk,
    input logic          rst_n,

    // Porta de escrita
    input logic          we,           // write enable
    input logic [$clog2(DEPTH)-1:0] waddr, // endereço de escrita
    input logic [WIDTH-1:0] wdata,     // dado de escrita

    // Duas portas de leitura combinacional
    input logic [$clog2(DEPTH)-1:0] raddr_a, // endereço leitura
    output logic [WIDTH-1:0] rdata_a,        // dado leitura A
    input logic [$clog2(DEPTH)-1:0] raddr_b, // endereço leitura
    output logic [WIDTH-1:0] rdata_b        // dado leitura B

```

);

// Array fixo: DEPTH elementos, cada um com WIDTH bits

logic [WIDTH-1:0] mem [0:DEPTH-1];

// Escrita síncrona + reset assíncrono

always_ff @(posedge clk or negedge rst_n) begin

if (!rst_n) begin

for (int i = 0; i < DEPTH; i++) begin

mem[i] <= '0;

end

end else if (we) begin

mem[waddr] <= wdata;

end

end

// Leituras combinacionais (assíncronas)

assign rdata_a = mem[raddr_a];

assign rdata_b = mem[raddr_b];


```
endmodule
```

O módulo `register_bank` concentra a **lógica do banco**. O array fixo `mem` representa a memória de registradores; o bloco `always_ff` implementa **escrita síncrona** (grava em `mem[waddr]` na borda de subida do clock quando `we=1`) e **reset assíncrono** (zera todos os registradores quando `rst_n=0`).

As leituras são **combinacionais**, mapeadas diretamente para `mem[raddr_a]` e `mem[raddr_b]`, permitindo observar alterações de endereço refletidas imediatamente nas saídas. A parametrização (`WIDTH`, `DEPTH`) facilita a reutilização e **`$clog2(DEPTH)`** dimensiona automaticamente os barramentos de endereço, reduzindo erros quando a profundidade muda.

tb.sv — testbench: escrita, leitura e verificação

```
module tb;
```

```
    localparam int WIDTH = 16;
```

```
    localparam int DEPTH = 8;
```

```
    localparam int AW    = $clog2(DEPTH);
```

```
    // Sinais
```

```
logic          clk, rst_n;

logic          we;

logic [AW-1:0]  waddr;

logic [WIDTH-1:0]  wdata;

logic [AW-1:0]  raddr_a, raddr_b;

logic [WIDTH-1:0]  rdata_a, rdata_b;
```

```
// DUT
```

```
top #(.WIDTH(WIDTH), .DEPTH(DEPTH)) dut (

    .clk    (clk),

    .rst_n  (rst_n),

    .we     (we),

    .waddr  (waddr),

    .wdata  (wdata),

    .raddr_a (raddr_a),

    .rdata_a (rdata_a),

    .raddr_b (raddr_b),

    .rdata_b (rdata_b)

);
```

```
// Clock 100 MHz (10 ns)

initial clk = 1'b0;

always #5 clk = ~clk;


// Waveform (EPWave/GTKWave)

initial begin

    $dumpfile("waveform.vcd");

    $dumpvars(0, tb);

end


// Tarefas auxiliares

task automatic write_reg(input logic [AW-1:0] addr, input logic
[WIDTH-1:0] data);

    begin

        @(negedge clk);

        we    = 1'b1;

        waddr = addr;

        wdata = data;

        @(negedge clk);

        we    = 1'b0;
```

```
end  
endtask
```

```
task automatic read_regs(input logic [AW-1:0] addr_a, input logic  
[AW-1:0] addr_b);
```

```
begin
```

```
  raddr_a = addr_a;
```

```
  raddr_b = addr_b;
```

```
  #1; // leitura combinacional estabiliza
```

```
  $display("t=%0t R[%0d]=0x%0h | R[%0d]=0x%0h",
```

```
    $time, addr_a, rdata_a, addr_b, rdata_b);
```

```
end
```

```
endtask
```

```
// Estímulos
```

```
initial begin
```

```
  // Reset inicial
```

```
  rst_n  = 1'b0;
```

```
  we     = 1'b0;
```

```
  waddr  = '0;
```



```
wdata = '0;
```

```
raddr_a = '0;
```

```
raddr_b = '0;
```

```
repeat (2) @(negedge clk);
```

```
rst_n = 1'b1;
```

```
// 1) Escreve padrão conhecido:  $R[i] = i * 3$ 
```

```
for (int i = 0; i < DEPTH; i++) begin
```

```
    write_reg(i[AW-1:0], WIDTH'(i*3));
```

```
end
```

```
// 2) Leituras paralelas (duas portas)
```

```
read_regs(0, 1);
```

```
read_regs(2, 5);
```

```
read_regs(7, 3);
```

```
// 3) Write-then-read no mesmo endereço
```

```
write_reg(4, 16'hABCD);
```

```
read_regs(4, 7); // antes da próxima borda útil de escrita
```

```
@(negedge clk);  
read_regs(4, 1); // após a borda, novo valor visível  
  
repeat (2) @(negedge clk);  
$finish;  
end  
  
endmodule
```

O testbench configura clock e reset, grava um **padrão conhecido** em todas as posições ($R[i] = i*3$) e realiza leituras **em paralelo** pelas duas portas, evidenciando o acesso simultâneo. As tarefas write_reg e read_regs organizam os estímulos: a escrita acontece de forma **síncrona** (na borda de clock subsequente com we=1), enquanto a leitura é **combinacional** (o dado aparece assim que o endereço muda).

O trecho de **write-then-read** no mesmo endereço ilustra a temporalidade: antes da borda útil, lê-se o valor antigo; após a borda, o valor atualizado. O par \$dumpfile/\$dumpvars habilita a inspeção de formas de onda no EPWave/GTKWave.

2.5. Fila de Impressão com Queue

A fila de impressão é um cenário clássico para explorar **estruturas dinâmicas** em SystemVerilog. Uma queue cresce ou encolhe em tempo de simulação conforme documentos são **enfileirados** (enviados para impressão) ou **desenfileirados** (impressos/retirados).

Para tornar o exemplo mais realista, cada “job” de impressão é descrito por uma struct com **nome do documento** e **número de páginas**. O módulo de design expõe **tarefas e funções** que permitem ao testbench manipular a fila por **chamadas hierárquicas** (enqueue, dequeue, peek, size, empty, total_pages), reproduzindo a interface de um spooler simples.

design.sv – top

```
`include "print_queue.sv"

module top;

    // O top apenas instancia o spooler de impressão.
    print_spooler u_spooler();

endmodule
```

Este arquivo integra o design. A diretiva ``include "print_queue.sv"` torna o módulo `print_spooler` visível para compilação a partir do `design.sv`. O top instancia um único spooler (`u_spooler`) sem portas, pois a interação com a fila acontecerá por **chamadas hierárquicas** a tarefas e funções do módulo (ex.: `dut.u_spooler.enqueue(...)`) a partir do testbench.

Essa abordagem simplifica o foco na **estrutura de dados** (fila) e nas **operações de alto nível**, sem overhead de sinais.

print_queue.sv — spooler com queue e operações

`// Spooler de impressão usando array estático`

```
module print_spooler;
```

```
    // Parâmetros da fila
```


```
    parameter MAX_JOBS = 100;
```

```
    // Arrays separados para nome e páginas (em vez de struct)
```

```
    string job_names[0:MAX_JOBS-1];
```

```
    int    job_pages[0:MAX_JOBS-1];
```

```
    // Controle da fila
```

```
int queue_size;
```

```
int front_ptr;
```

```
int rear_ptr;
```

```
// Inicialização
```

```
initial begin
```

```
    queue_size = 0;
```

```
    front_ptr = 0;
```

```
    rear_ptr = 0;
```

```
// Inicializar arrays
```

```
for (int i = 0; i < MAX_JOBS; i++) begin
```

```
    job_names[i] = "";
```

```
    job_pages[i] = 0;
```

```
end
```

```
end
```

```
// Insere no fim da fila
```

```
task automatic enqueue(input string name, input int pages);
```

```
    if (queue_size >= MAX_JOBS) begin
```

```
$display("[ENQUEUE] ERRO: Fila cheia! Não foi possível  
adicionar '%s'", name);
```

```
end else begin
```

```
    job_names[rear_ptr] = name;
```

```
    job_pages[rear_ptr] = pages;
```

```
    rear_ptr = (rear_ptr + 1) % MAX_JOBS;
```

```
    queue_size = queue_size + 1;
```

```
$display("[ENQUEUE] '%s' (%0d pág). Tamanho atual: %0d",  
name, pages, queue_size);
```

```
end
```

```
endtask
```

```
// Remove do início da fila (se houver)
```

```
task automatic dequeue(output string name, output int pages,  
output bit ok);
```

```
    if (queue_size == 0) begin
```

```
        ok = 0;
```

```
        name = "";
```

```
        pages = 0;
```

```
$display("[DEQUEUE] Fila vazia.");
```

```
end else begin
    ok = 1;
    name = job_names[front_ptr];
    pages = job_pages[front_ptr];

    // Limpar posição
    job_names[front_ptr] = "";
    job_pages[front_ptr] = 0;

    front_ptr = (front_ptr + 1) % MAX_JOBS;
    queue_size = queue_size - 1;

    $display("[DEQUEUE] '%s' (%0d pág). Tamanho atual: %0d",
name, pages, queue_size);

    end
endtask

// Olha o próximo da fila sem remover
task automatic peek(output string name, output int pages, output
bit ok);
```

```
if (queue_size == 0) begin
    ok = 0;
    name = "";
    pages = 0;
    $display("[PEEK] Fila vazia.");
end else begin
    ok = 1;
    name = job_names[front_ptr];
    pages = job_pages[front_ptr];
    $display("[PEEK] Próximo: '%s' (%0d pág).", name, pages);
end
endtask
```

```
// Quantidade de jobs na fila
```

```
function automatic int size();
```

```
    return queue_size;
```

```
endfunction
```

```
// Fila está vazia?
```

```
function automatic bit empty();
```



```
    return (queue_size == 0);
endfunction

// Soma total de páginas pendentes
function automatic int total_pages();

    int sum = 0;

    int idx = front_ptr;

    for (int i = 0; i < queue_size; i++) begin

        sum = sum + job_pages[idx];

        idx = (idx + 1) % MAX_JOBS;

    end

    return sum;

endfunction

// Imprime um snapshot do estado da fila
task automatic print_status();

    int idx;
```

```

$display("----- STATUS DA FILA -----");

$display("Jobs na fila: %0d | Páginas totais: %0d", size(),
total_pages());

if (queue_size == 0) begin
    $display(" (vazia)");
end else begin
    idx = front_ptr;

    for (int i = 0; i < queue_size; i++) begin
        $display(" [%0d] '%s' (%0d pág)", i, job_names[idx],
job_pages[idx]);

        idx = (idx + 1) % MAX_JOBS;
    end
end

$display("-----");

endtask

endmodule

```

O módulo **print_spooler** mantém duas estruturas paralelas: um array de string para os nomes dos documentos e um array de int para o número de páginas. O controle da fila é feito por variáveis de índice e tamanho, simulando o comportamento de uma fila circular. As operações principais são implementadas como tarefas e funções: **enqueue** insere no final, **dequeue** remove do início e retorna também um indicador ok, **peek** permite inspecionar o próximo sem remover, **size** e **empty** consultam o estado, **total_pages** acumula o número de páginas pendentes e **print_status** gera um snapshot amigável da fila.

Esse modelo mostra como operações típicas de um spooler podem ser reproduzidas em ambiente de simulação, com foco em verificação e aprendizagem, e não em síntese direta para hardware.

tb.sv — testbench: enfileirar, inspecionar e imprimir jobs

```
module tb;
```

```
    // Instância do design top
```

```
    top dut();
```

```
    // Variáveis para receber dados do dequeue/peek
```

```
    string job_name;
```

```
    int job_pages;
```

```
    bit ok;
```

// Geração simples de "tempo" e checkpoints de log

initial begin

\$dumpfile("waveform.vcd");

\$dumpvars(0, tb);

// Início: fila vazia

dut.u_spooler.print_status();

// 1) Enfileira três documentos

dut.u_spooler.enqueue("Relatorio_Projeto.pdf", 12);

dut.u_spooler.enqueue("Apresentacao.pptx", 25);

dut.u_spooler.enqueue("Planilha_Custos.xlsx", 7);

// Snapshot do estado

dut.u_spooler.print_status();

// 2) Espiar quem é o próximo

dut.u_spooler.peek(job_name, job_pages, ok);

// 3) Imprimir (dequeue) dois documentos


```
dut.u_spooler.dequeue(job_name, job_pages, ok);
dut.u_spooler.dequeue(job_name, job_pages, ok);

// Estado após duas remoções
dut.u_spooler.print_status();

// 4) Inserir mais um job e checar métricas
dut.u_spooler.enqueue("Fotos_Evento.zip", 40);
$display("Size=%0d | Empty=%0b | TotalPages=%0d",
        dut.u_spooler.size(), dut.u_spooler.empty(),
dut.u_spooler.total_pages());

// 5) Esvaziar completamente a fila
while (!dut.u_spooler.empty()) begin
    dut.u_spooler.dequeue(job_name, job_pages, ok);
end
dut.u_spooler.print_status();

// 6) Testar remoção/peek com fila vazia
dut.u_spooler.dequeue(job_name, job_pages, ok);
```

```

dut.u_spooler.peek(job_name, job_pages, ok);

// 7) Teste adicional: encher a fila
$display("\n--- TESTE DE CAPACIDADE ---");

for (int i = 1; i <= 5; i++) begin
    $sformatf(job_name, "Job_%0d.pdf", i);
    dut.u_spooler.enqueue(job_name, i * 3);
end

dut.u_spooler.print_status();

// Encerrar simulação
#10 $finish;

end

endmodule

```

O testbench interage com o spooler por chamadas hierárquicas (por exemplo, `dut.u_spooler.enqueue(...)`), seguindo um fluxo natural: enfileira três documentos, inspeciona o próximo com **peek**, remove dois com **dequeue**, insere mais um e consulta métricas como **size**, **empty** e **total_pages**. Em seguida, a fila é

esvaziada completamente e são exercitados os casos de borda ao tentar remover ou inspecionar quando não há elementos, com mensagens apropriadas no log.

Por fim, há um teste adicional de capacidade, enfileirando vários jobs em sequência para verificar o funcionamento da estrutura circular.

2.6. Representação de Pixels com struct

Este exemplo modela uma imagem **RGB** por meio de três matrizes bidimensionais de **8 bits**, correspondentes aos canais **R**, **G** e **B**. Cada posição da matriz representa um pixel, e funções auxiliares permitem empacotar os três canais em um único valor de **24 bits (0xRRGGBB)** ou desempacotar esse valor em componentes individuais.

O módulo de design disponibiliza operações para preencher a imagem inteira, definir ou ler um pixel específico, inverter as cores e converter para tons de cinza. O **testbench** conduz o fluxo completo e imprime a imagem no console como uma grade de valores no formato **0xRRGGBB**.

design.sv – top

```
`include "pixel_image.sv"
```

```
module top;
```

```
// Instancia a "imagem" com parâmetros padrão (largura x altura)
```

```
pixel_image #(.W(4), .H(3)) u_img();
```

```
endmodule
```

O **design.sv** apenas integra o projeto trazendo o módulo **pixel_image** por meio do include "pixel_image.sv" e o instancia como **u_img**. A interação com a imagem é feita pelo testbench por chamadas hierárquicas às tarefas e funções do módulo, mantendo o foco nas operações sobre os arrays de canais RGB, sem a necessidade de sinais externos.

pixel_image.sv — matriz de pixels com operações

```
// Módulo de imagem usando arrays separados para RGB
```

```
module pixel_image #(parameter int W = 4, H = 3);
```

```
// Arrays separados para canais RGB (substituindo struct packed)
```

```
logic [7:0] img_r [0:H-1][0:W-1]; // Canal Red
```

```
logic [7:0] img_g [0:H-1][0:W-1]; // Canal Green
```

```
logic [7:0] img_b [0:H-1][0:W-1]; // Canal Blue
```

```
// Constrói um pixel a partir de canais (retorna valor empacotado)
```

```
function automatic logic [23:0] mk_pixel(input logic [7:0] r, g, b);
```

```
    return {r, g, b};
```



```
endfunction
```

```
// Converte para 24 bits (0xRRGGBB)
```

```
function automatic logic [23:0] pack24(input logic [7:0] r, g, b);
```

```
    return {r, g, b};
```

```
endfunction
```

```
// Constrói pixel a partir de 24 bits (0xRRGGBB) - retorna apenas R
```

```
function automatic logic [7:0] unpack24_r(input logic [23:0] x);
```

```
    return x[23:16];
```

```
endfunction
```

```
// Extraí canal G de 24 bits
```

```
function automatic logic [7:0] unpack24_g(input logic [23:0] x);
```

```
    return x[15:8];
```

```
endfunction
```

```
// Extraí canal B de 24 bits
```

```
function automatic logic [7:0] unpack24_b(input logic [23:0] x);
```

```
    return x[7:0];
```

```
endfunction
```

```
// Preenche a imagem inteira com uma cor
```

```
task automatic fill(input logic [7:0] r, g, b);
```

```
  for (int y = 0; y < H; y++) begin
```

```
    for (int x = 0; x < W; x++) begin
```

```
      img_r[y][x] = r;
```

```
      img_g[y][x] = g;
```

```
      img_b[y][x] = b;
```

```
    end
```

```
  end
```

```
endtask
```

```
// Define um pixel (com checagem simples de limites)
```

```
task automatic set_pixel(input int x, y, input logic [7:0] r, g, b);
```

```
  if (x >= 0 && x < W && y >= 0 && y < H) begin
```

```
    img_r[y][x] = r;
```

```
    img_g[y][x] = g;
```

```
    img_b[y][x] = b;
```

```
  end else begin
```

```

$display("[WARN] set_pixel fora da área: x=%0d y=%0d", x, y);
end
endtask

// Lê um pixel (retorna 'ok' = 0 se fora da área)
task automatic get_pixel(input int x, y, output logic [7:0] r, g, b,
output bit ok);

    if (x >= 0 && x < W && y >= 0 && y < H) begin

        r = img_r[y][x];

        g = img_g[y][x];

        b = img_b[y][x];

        ok = 1;

    end else begin

        r = 8'h00;

        g = 8'h00;

        b = 8'h00;

        ok = 0;

    end
endtask

```

// Inverte cores de toda a imagem (negativo)

task automatic invert();

for (int y = 0; y < H; y++) begin

for (int x = 0; x < W; x++) begin

img_r[y][x] = ~img_r[y][x];

img_g[y][x] = ~img_g[y][x];

img_b[y][x] = ~img_b[y][x];

end

end

endtask

// Converte para tom de cinza por média simples $(r+g+b)/3$

task automatic to_grayscale();

int gray;

for (int y = 0; y < H; y++) begin

for (int x = 0; x < W; x++) begin

gray = (img_r[y][x] + img_g[y][x] + img_b[y][x]) / 3;

img_r[y][x] = gray[7:0];

img_g[y][x] = gray[7:0];

img_b[y][x] = gray[7:0];

```

    end
end
endtask

// Imprime a imagem no console em formato 0xRRGGBB
task automatic print_image(string title);
    if (title == "") title = "IMAGE";
    $display("---- %s (%0dx%0d) ----", title, W, H);
    for (int y = 0; y < H; y++) begin
        string line = "";
        for (int x = 0; x < W; x++) begin
            line = {line, $sprintf(" 0x%06h", pack24(img_r[y][x],
img_g[y][x], img_b[y][x]))};
        end
        $display("%s", line);
    end
    $display("-----");
endtask

endmodule

```


O módulo `pixel_image` define `pixel_t` como struct packed com três campos de 8 bits (`r`, `g`, `b`). Por ser *packed*, a struct é fisicamente contígua e pode ser **empacotada** e **desempacotada** com $\{p.r, p.g, p.b\} \rightleftharpoons [23:0]$, graças às funções `pack24` e `unpack24`.

A imagem é uma matriz `img[H][W]` de `pixel_t`. As tarefas `fill`, `set_pixel` e `get_pixel` implementam operações básicas de edição; `invert` gera o negativo da imagem; `to_grayscale` aplica um tom de cinza simples pela média. A tarefa `print_image` percorre a matriz e imprime cada pixel como `0xRRGGBB`, facilitando a inspeção no console e mostrando o benefício prático de uma struct packed (fácil de manipular por campos e como vetor).

tb.sv — testbench:

```
module tb;
```

```
    top dut();
```

```
// Todas as variáveis declaradas no topo (compatível com Icarus)
```

```
logic [7:0] r, g, b;
```

```
logic [7:0] mag_r, mag_g, mag_b;
```

```
logic [23:0] magenta24;
```

```
bit ok;
```

```
initial begin
```

```
$dumpfile("waveform.vcd");  
$dumpvars(0, tb);  
  
// 1) Preenche a imagem com azul  
dut.u_img.fill(8'h00, 8'h00, 8'hFF); // BLUE  
dut.u_img.print_image("FILL BLUE");  
  
// 2) Define alguns pixels coloridos  
dut.u_img.set_pixel(0, 0, 8'hFF, 8'h00, 8'h00); // RED  
dut.u_img.set_pixel(1, 0, 8'h00, 8'hFF, 8'h00); // GREEN  
dut.u_img.set_pixel(2, 0, 8'hFF, 8'hFF, 8'hFF); // WHITE  
dut.u_img.set_pixel(3, 2, 8'h80, 8'h80, 8'h00); // amarelo escuro  
dut.u_img.print_image("AFTER SET_PIXEL");  
  
// 3) Lê um pixel específico e mostra campos individuais  
dut.u_img.get_pixel(2, 0, r, g, b, ok);  
  
if (ok) $display("Pixel(2,0): R=0x%02h G=0x%02h B=0x%02h  
(packed=0x%06h)", r, g, b, dut.u_img.pack24(r, g, b));  
  
// 4) Inverte todas as cores
```

```
dut.u_img.invert();  
dut.u_img.print_image("AFTER INVERT");  
  
// 5) Converte para tons de cinza  
dut.u_img.to_grayscale();  
dut.u_img.print_image("AFTER GRAYSCALE");  
  
// 6) Demonstra empacotamento/desempacotamento 24b ->  
componentes RGB  
  
magenta24 = 24'hFF00FF;  
mag_r = magenta24[23:16]; // Canal R  
mag_g = magenta24[15:8];  // Canal G  
mag_b = magenta24[7:0];   // Canal B  
dut.u_img.set_pixel(0, 2, mag_r, mag_g, mag_b);  
dut.u_img.print_image("AFTER UNPACK24 (MAGENTA @ 0,2)");  
  
#10 $finish;  
  
end  
  
endmodule
```

O **testbench** exercita as operações principais: inicia preenchendo a imagem inteira com azul, faz edições pontuais em alguns pixels (definindo **RED**, **GREEN**, **WHITE** e um amarelo escuro), lê um pixel específico para exibir seus canais individuais e o valor empacotado, aplica inversão global das cores, converte a imagem para tons de cinza e demonstra o uso do empacotamento/desempacotamento de valores **0xRRGGBB** para componentes RGB.

A impressão em grade com **print_image** facilita a visualização das mudanças em cada etapa. As chamadas hierárquicas (**dut.u_img.***) reforçam que o foco do exemplo está na manipulação dos dados da imagem em ambiente de simulação, e não em sinais de I/O de hardware.

2.7. Semáforo com Enum (FSM simples)

Este exemplo descreve um semáforo veicular controlado por uma **máquina de estados finitos (FSM)** do tipo Moore. Os estados são definidos com enum para dar **nomes semânticos** às fases do sinal (**S_RED**, **S_GREEN**, **S_YELLOW**), melhorando a legibilidade e reduzindo erros.

Cada estado permanece ativo por um **tempo parametrizável** em ciclos de clock, controlado por um contador interno. Ao término do tempo, a FSM avança de estado. O design expõe as saídas de lâmpadas (**red**, **yellow**, **green**) e o **estado atual**

para observação. O testbench executa a sequência completa, imprime as mudanças e registra as ondas.

design.sv – top

```
`include "traffic_light.sv"

module top;

    // Instancia o semáforo com tempos padrão (em ciclos de clock)
    // Ex.: GREEN=12, YELLOW=4, RED=10 — ajuste livre no TB se
    // desejar

    traffic_light #(
        .T_GREEN (12),
        .T_YELLOW(4),
        .T_RED   (10)
    ) u_fsm (
        .clk   (),
        .rst_n (),
        .red   (),
        .yellow(),
        .green (),
        .state ()
    );
endmodule
```



```
);  
endmodule
```

O arquivo **design.sv** integra o sistema trazendo o módulo **traffic_light** por meio do include "traffic_light.sv" e instanciando-o como **u_fsm**. Os tempos de cada fase (**verde**, **amarelo** e **vermelho**) são parametrizados em ciclos de clock, permitindo ajustar facilmente a duração de cada estado. No **testbench**, a interação ocorre pela conexão explícita dos sinais de entrada e saída à instância, já que o top apenas encapsula a FSM.

traffic_light.sv — FSM Moore com enum e tempos parametrizados

// FSM de semáforo veicular (Moore) com tempos parametrizados

```
module traffic_light #(  
    parameter int T_GREEN  = 12,  
    parameter int T_YELLOW = 4,  
    parameter int T_RED    = 10  
)(  
    input  logic clk,  
    input  logic rst_n,  
    output logic red,
```

```
output logic yellow,  
output logic green,  
output logic [1:0] state  
);
```

```
// Estados definidos como parâmetros locais (em vez de enum)
```

```
localparam logic [1:0] S_RED    = 2'b00;
```

```
localparam logic [1:0] S_GREEN  = 2'b01;
```

```
localparam logic [1:0] S_YELLOW = 2'b10;
```

```
logic [1:0] curr, next;
```

```
int unsigned tick; // contador de tempo dentro do estado
```

```
// Saídas Moore: dependem apenas do estado
```

```
always_comb begin
```

```
    red    = (curr == S_RED);
```

```
    yellow = (curr == S_YELLOW);
```

```
    green  = (curr == S_GREEN);
```

```
end
```

// Próximo estado e controle de término de tempo

// Quando 'tick' atinge ($T_* - 1$), avança para o próximo estado

always_comb begin

 next = curr;

 case (curr)

 S_GREEN: next = (tick == T_GREEN - 1) ? S_YELLOW :
S_GREEN;

 S_YELLOW: next = (tick == T_YELLOW - 1) ? S_RED :
S_YELLOW;

 S_RED: next = (tick == T_RED - 1) ? S_GREEN : S_RED;

 default: next = S_RED;

 endcase

end

// Estado e temporização (sequencial)

always_ff @(posedge clk or negedge rst_n) begin

 if (!rst_n) begin

 curr <= S_RED;

 tick <= 0;

 end else begin

```
if (next != curr) begin
    curr <= next;

    tick <= 0;          // ao mudar de estado, zera o contador
end else begin
    tick <= tick + 1;    // segue contando dentro do mesmo estado
end
end
end

// Exporta o estado atual para observação externa
assign state = curr;

endmodule
```

O módulo **traffic_light** implementa uma FSM do tipo **Moore** com três estados representados por parâmetros locais (**S_RED**, **S_GREEN**, **S_YELLOW**). As saídas (**red**, **yellow**, **green**) dependem apenas do estado atual, o que reduz glitches e facilita a verificação. Um contador **tick** acompanha quantos ciclos de clock já se passaram em cada estado; quando atinge o valor máximo definido pelos parâmetros (**T_GREEN**, **T_YELLOW**, **T_RED**), a lógica de próximo estado (**next**) avança para a fase seguinte e o contador é zerado.

A separação entre **always_ff** (parte sequencial) e **always_comb** (parte combinacional) segue as boas práticas de modelagem em SystemVerilog.

tb.sv — testbench: clock/reset, observação de estados e tempos

```
module tb;
```

```
// Sinais do DUT
```

```
logic clk, rst_n;
```

```
logic red, yellow, green;
```

```
logic [1:0] state;
```

```
// Estados definidos localmente (mesmos valores do módulo)
```

```
localparam logic [1:0] S_RED    = 2'b00;
```

```
localparam logic [1:0] S_GREEN  = 2'b01;
```

```
localparam logic [1:0] S_YELLOW = 2'b10;
```

```
// Instancia o design integrador (top) e
```

```
// conecta explicitamente os sinais à instância u_fsm
```

```
top dut();
```


// Conexões explícitas aos pinos não amarrados do 'top'

// (o 'top' apenas instanciou a FSM; aqui ligamos sinais)

```
assign dut.u_fsm.clk    = clk;
```

```
assign dut.u_fsm.rst_n  = rst_n;
```

```
assign red              = dut.u_fsm.red;
```

```
assign yellow           = dut.u_fsm.yellow;
```

```
assign green            = dut.u_fsm.green;
```

```
assign state            = dut.u_fsm.state;
```

// Geração de clock (100 MHz -> período 10 ns)

```
initial clk = 1'b0;
```

```
always #5 clk = ~clk;
```

// Waveform para EPWave/GTKWave

```
initial begin
```

```
    $dumpfile("waveform.vcd");
```

```
    $dumpvars(0, tb);
```

```
end
```

// Função utilitária para imprimir o nome do estado

```
function string state_name(input logic [1:0] s);
```

```
  case (s)
```

```
    S_RED:   return "RED";
```

```
    S_GREEN: return "GREEN";
```

```
    S_YELLOW: return "YELLOW";
```

```
    default: return "UNKNOWN";
```

```
  endcase
```

```
endfunction
```

```
// Monitora transições de estado e tempos
```

```
logic [1:0] last_state;
```

```
int      dwell;
```

```
initial begin
```

```
  // Reset inicial
```

```
  rst_n    = 1'b0;
```

```
  last_state = 2'bxx;
```

```
  dwell     = 0;
```

```
  repeat (3) @(negedge clk);
```

```
  rst_n = 1'b1;
```

```

// Roda por alguns ciclos suficientes para observar várias voltas
repeat (60) begin
    @(posedge clk);
    if (state !== last_state) begin
        // Ao detectar mudança, relata quanto tempo o estado anterior
        durou
        if (last_state !== 2'bxx) begin
            $display("t=%0t leave %-6s after %0d cycles",
                    $time, state_name(last_state), dwell);
        end
        $display("t=%0t enter %-6s (R=%0b Y=%0b G=%0b)",
                $time, state_name(state), red, yellow, green);
        last_state = state;
        dwell      = 1;
    end else begin
        dwell++;
    end
end
end
$finish;

```

```
end  
endmodule
```

O **testbench** conecta o clock e o reset à instância **u_fsm**, captura as saídas das lâmpadas (**red, yellow, green**) e o estado atual, e executa a simulação por tempo suficiente para observar múltiplos ciclos completos (**VERDE → AMARELO → VERMELHO → VERDE...**). A cada transição, imprime o nome da fase iniciada, o tempo de permanência no estado anterior (em ciclos) e os valores das saídas no instante de entrada do novo estado.

O arquivo de ondas (**waveform.vcd**) permite confirmar visualmente a sequência e a duração de cada fase, enquanto a função auxiliar **state_name** torna os relatórios mais legíveis ao traduzir os códigos binários dos estados em nomes textuais.

2.8. Conclusão

Ao longo deste material foi possível explorar de forma progressiva os principais recursos do **SystemVerilog** aplicados ao desenvolvimento de projetos digitais, desde conceitos básicos até estruturas mais avançadas. Foram apresentados exemplos que demonstraram a evolução natural da linguagem em relação ao Verilog clássico, com destaque para o uso de **logic** no lugar de **reg** e **wire**, a criação de módulos hierárquicos, e a importância dos **testbenches** na verificação funcional.

Com base em atividades práticas, analisamos como arrays fixos podem representar bancos de registradores, como filas dinâmicas (queue) podem modelar sistemas reais de espera, e como estruturas (struct) permitem agrupar informações de maneira clara, como no caso da representação de pixels RGB. Além disso, o uso de enum para modelagem de máquinas de estados mostrou a relevância de recursos de alto nível na descrição de circuitos de controle.

Esses exemplos evidenciam que o SystemVerilog não é apenas uma evolução sintática, mas uma ferramenta poderosa para aproximar a **descrição de hardware** da **modelagem de sistemas complexos**. Ele permite trabalhar de forma mais expressiva, reduzindo erros e aumentando a produtividade tanto em projetos de **síntese para FPGA/ASIC** quanto em **simulação e verificação**.

Assim, conclui-se que dominar os recursos do SystemVerilog é fundamental para quem deseja projetar sistemas digitais modernos, combinando clareza na descrição, robustez na verificação e escalabilidade na construção de arquiteturas hierárquicas. O conhecimento adquirido aqui serve como base para avançar em projetos mais complexos, como processadores, controladores e sistemas embarcados, além de preparar o terreno para metodologias profissionais de verificação, como UVM (Universal Verification Methodology).

Referências

DOULOS. EDA Playground. Disponível em:
<https://www.edaplayground.com/>. Acesso em: 27 ago. 2025.

