DISCIPLINA EA701
Introdução aos Sistemas Embarcados

ROTEIRO 12: Integração de HDL em Sistemas Embarcados: Da Teoria à Prática.
Níveis de Abstração de *Hardware*, Dispositivos Lógicos Programáveis (PLD), Linguagem de Descrição de *Hardware* (VHDL), Modelagem Gráfica Abstrata UML, Modelagem Comportamental (Sequencial) com Máquinas de Estados (FSM) e Máquinas de Estados e Caminho de Dados (FSMD).

Profs. Antonio A. F. Quevedo e Wu Shin-Ting

FEEC / UNICAMP

Revisado e modificado em junho de 2025 por Ting com auxílio do Chatgpt
Revisado em novembro de 2024

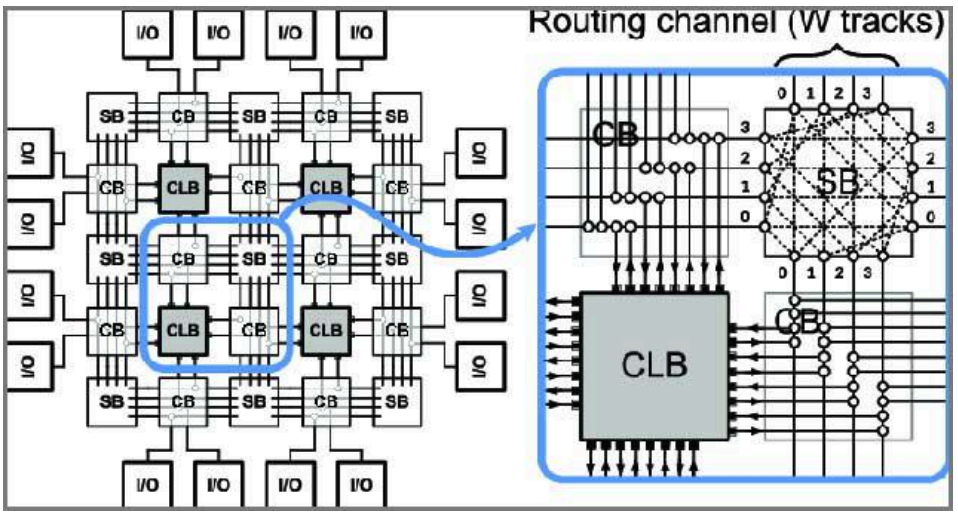
INTRODUÇÃO

No universo dos sistemas embarcados, a intersecção entre *hardware* e *software* é fundamental para o sucesso de projetos inovadores. Nesse cenário, a programabilidade dos microcontroladores se destaca como um diferencial, permitindo personalizar funcionalidades do *hardware* por meio de *software* como vimos nos Roteiros anteriores, adaptando dispositivos a uma ampla gama de aplicações com flexibilidade e agilidade. Essa capacidade de reconfiguração dinâmica torna possível implementar algoritmos de controle, comunicação e processamento de sinais diretamente no dispositivo, acelerando o desenvolvimento e reduzindo custos com prototipagem.

No entanto, há mecanismos eletrônicos que permanecem intrinsecamente dependentes de soluções a nível de *hardware*, mesmo no estado-da-arte da tecnologia. Exemplos notáveis incluem conexões do tipo *wired-and*, comumente utilizadas em barramentos de comunicação

como o I2C, como vimos no Roteiro 10, e a lógica de arbitragem baseada em *bits* dominantes e recessivos, típica do protocolo CAN, apresentada no Roteiro 11. Tais técnicas exploram características elétricas dos circuitos, como impedância e níveis de tensão, que não podem ser emuladas de forma confiável apenas via *software*, exigindo um projeto de *hardware* específico e dedicado. Outro exemplo é o combate a ruídos eletromagnéticos (em inglês, *Electromagnetic Interference* - EMI, *Radio Frequency Interference* - RFI) em sinais de entrada dos microcontroladores, mostrados no Roteiro 8. Como esses ruídos são fenômenos físicos (ondas eletromagnéticas ou correntes parasitas que se acoplam aos sinais), sua atenuação exige intervenção no nível físico do sinal, sendo amplamente tratada por *hardware* na maioria dos sistemas eletrônicos.

Nesse contexto, a linguagem HDL (do inglês *Hardware Description Language*) emerge como uma ferramenta essencial para a descrição e implementação de circuitos digitais em dispositivos de lógica programável do estado da arte, como FPGAs e ASICs. Dispositivos de Lógica Programável (em inglês, *Programmable Logic Devices* - PLDs) são componentes eletrônicos que permitem aos *designers* configurar sua lógica interna, em vez de serem fabricados para uma função fixa específica. Isso oferece uma incrível flexibilidade no desenvolvimento de *hardware*, permitindo que a funcionalidade de um *chip* seja definida ou alterada por meio de "programação". Os PLDs modernos, como as FPGAs (do inglês *Field-Programmable Gate Arrays*), consistem em uma matriz de blocos lógicos configuráveis e interconexões programáveis, o que lhes permite implementar virtualmente qualquer circuito digital. Já os ASICs (do inglês *Application-Specific Integrated Circuits*) são circuitos integrados projetados para uma aplicação específica e são fabricados sob medida para uma função particular, oferecendo o máximo em desempenho e eficiência, mas com custos de desenvolvimento e fabricação muito mais altos e sem a flexibilidade de reprogramação. A figura que segue ilustra a lógica interna de um FPGAs. Ela é composta por um conjunto de blocos lógicos reconfiguráveis (em inglês, *Configurable Logic Block* - CLB), interconectados por uma rede programável de roteamento formada pelos blocos de conexão (em inglês, Connection Block - CB) e blocos de chaveamento (em inglês, *Switch Block* - SB) elementos de memória. Essa configuração é definida e carregada no dispositivo por meio de uma HDL, como VHDL (do inglês ***V**ery high speed integrated circuit **H**ardware **D**escription **L**anguage*) ou Verilog.



Fonte: WELLPCB

Assim, a HDL abre as portas para o futuro da engenharia digital, onde o conceito de "desenhar" um circuito vai além da simples montagem de componentes físicos e esquemáticos. Ela oferece novas perspectivas no *design* e na implementação de sistemas eletrônicos avançados, permitindo que engenheiros descrevam o comportamento e a estrutura dos PLDs de forma abstrata, facilitando o *design*, a verificação e a síntese para diferentes plataformas de *hardware*. A capacidade de modelar o comportamento do *hardware* de forma precisa permite que engenheiros e desenvolvedores criem soluções que não apenas atendam aos requisitos funcionais, mas que também sejam otimizadas em termos de desempenho, eficiência e compatibilidade elétrica, inclusive para lidar com essas exigências que só podem ser resolvidas no domínio físico dos circuitos. Ao longo deste semestre, vimos que, com a crescente complexidade dos sistemas embarcados, especialmente através do uso de microcontroladores avançados como o STM32, a integração entre *software* e *hardware* se torna cada vez mais complementar. Enquanto o STM32 oferece recursos robustos de processamento, temporização e controle periférico, o uso de HDL permite aos projetistas definir e implementar, de forma altamente personalizada, funções específicas de *hardware*, como interfaces de comunicação e controle de sinais críticos.

Essa abordagem integrada possibilita a maximização das capacidades do sistema, explorando o melhor dos dois mundos: a flexibilidade do *software* e a precisão do *hardware*. Assim, é possível desenvolver soluções embarcadas cada vez mais robustas, eficientes e adaptadas às exigências do ambiente real. Neste roteiro, vamos apresentar uma visão introdutória sobre o potencial da modelagem de *hardware* e sua sinergia com a programação de microcontroladores. Nosso objetivo é explorar a perspectiva do **co-projeto**, usando exemplos práticos e discutindo aplicações reais. Queremos destacar como a descrição de *hardware* se torna um elemento cada vez mais relevante no desenvolvimento de sistemas embarcados robustos e eficientes.

Este roteiro guiará você por uma introdução ao co-projeto de sistemas embarcados, destacando a integração entre o design de hardware e software para alcançar soluções mais

eficientes e otimizadas. Começaremos com os níveis de abstração de *hardware*, desde o nível de portas lógicas até o nível de sistema, estabelecendo uma base sólida para compreender a hierarquia do projeto digital. Em seguida, apresentaremos os dispositivos lógicos programáveis (PLDs), discutindo suas características, aplicações e papéis no desenvolvimento de *hardware* configurável. A seguir, introduziremos a linguagem VHDL, explorando sua sintaxe e uso para a descrição comportamental e estrutural de circuitos digitais, além de sua íntima relação com máquinas de estados. Para auxiliar na modelagem e na comunicação entre equipes multidisciplinares, abordaremos a UML (do inglês *Unified Modeling Language*) como ferramenta complementar ao projeto de *hardware*. Daremos continuidade com o estudo do FSDM (do inglês *Finite State Machine Design Method*), uma abordagem fundamental para o controle sequencial em circuitos digitais, especialmente em sistemas embarcados.

Devido a limitações de infraestrutura, nossos projetos de *hardware* se restringirão a simulações. No entanto, com o conhecimento que você adquiriu nos roteiros anteriores, essas simulações serão suficientes pra demonstrar que os sinais físicos gerados, como os enviados pelo microcontrolador e analisados por um analisador lógico, são integráveis no nível de simulação.

PROJETOS-EXEMPLO

Esta seção apresenta projetos-exemplo que combinam a modelagem gráfica de sistemas digitais utilizando UML (do inglês, *Unified Modeling Language*) e a descrição de *hardware* com VHDL. A abordagem explora como UML pode ser empregada para documentar e organizar funcionalmente o *design* de circuitos digitais, enquanto o VHDL traduz essas especificações em implementações sintetizáveis para dispositivos PLDs, como FPGAs ou ASICs. Além disso, os projetos são desenvolvidos e testados em um ambiente de simulação *online*, como o EDA Playground, permitindo validação prática e aprendizado interativo.

Projeto 1: Usando o ambiente EDA Playround e simulando um circuito combinacional

Em vez de se preocupar com a construção física de circuitos utilizando componentes discretos como transistores, portas lógicas e resistores, já imaginou poder sintetizar um controlador dedicado, como um conversor ADC ou um periférico UART, utilizando apenas uma linguagem que descreve o comportamento lógico e funcional do circuito? E, a partir dessa descrição, deixar que um sintetizador transforme essa representação em um circuito físico real? Este projeto combinacional de um **decodificador 3 para 8** tem como objetivo apresentar o VHDL como uma ferramenta para descrever, simular e sintetizar circuitos digitais (*hardware*) em dispositivos lógicos programáveis, como FPGAs.

É importante destacar que a descrição feita em VHDL não se limita a um modelo teórico: ela é convertida em um circuito combinacional real, com comportamento físico equivalente ao de um circuito construído com portas lógicas e conexões entre elas. Assim, quando é especificada uma rota da entrada até a saída, o sinal percorre todos os caminhos possíveis, como água fluindo em uma rede de canalização, propagando-se pelos fios e portas internas do dispositivo. Essa propagação de sinais não ocorre instantaneamente: existe um tempo de propagação associado a cada estágio do circuito, que corresponde ao intervalo necessário para que uma mudança na entrada se reflita na saída. Esse fator é crucial no projeto digital, pois influencia diretamente o desempenho do circuito e deve ser considerado tanto na simulação quanto na síntese. Em sistemas mais complexos, o tempo de propagação afeta o *timing* global e pode limitar a frequência máxima de operação do sistema. Essa natureza física e paralela dos circuitos se reflete diretamente na forma como o código VHDL deve ser interpretado. **Diferentemente das linguagens de programação tradicionais, onde a execução segue uma ordem sequencial de comandos**, em uma linguagem de descrição de *hardware*, **as instruções não são executadas em sequência, mas representam estruturas que operam simultaneamente**. Cada declaração no código HDL descreve um componente ou comportamento que será sintetizado como parte de um sistema de *hardware* operando em paralelo. Assim, a síntese do sistema a partir do código HDL gera uma arquitetura composta por **módulos paralelos e interconectados**, não uma linha de execução sequencial.

Esse comportamento evidencia uma das diferenças fundamentais entre a linguagem de programação de um microcontrolador e uma linguagem de descrição de *hardware* como o VHDL. Ao programar um microcontrolador, os sinais e caminhos de execução já estão predefinidos pela arquitetura interna do *chip*: o programador atua sobre registradores, periféricos e instruções de controle de fluxo, sem alterar a estrutura física do *hardware*. Por outro lado, ao usar uma linguagem de descrição de *hardware*, o desenvolvedor é responsável por definir os próprios sinais, conexões e caminhos lógicos que constituirão fisicamente o circuito. Isso implica que uma especificação incorreta em VHDL pode resultar em comportamentos de *hardware* totalmente erráticos ou até perigosos, uma vez que o erro está na própria definição estrutural do circuito — não apenas na lógica de controle. Diferentemente de *bugs* em *software*, que geralmente se manifestam como falhas previsíveis ou reinicializações, erros em *hardware* descrito incorretamente podem gerar conflitos elétricos, falhas de sincronização ou resultados lógicos imprevisíveis, com impactos potencialmente mais severos.

O ***EDA Playground*** é uma plataforma *online* gratuita para desenvolvimento, simulação e compartilhamento de códigos relacionados a *design* e verificação de circuitos digitais, especialmente voltados para a área de *Electronic Design Automation* (EDA). A plataforma é amplamente utilizada para experimentar linguagens de descrição de *hardware* como *SystemVerilog*, *Verilog*, e *VHDL*. Com o EDA Playground, é possível escrever e simular códigos diretamente no navegador, sem a necessidade de instalar ferramentas locais.

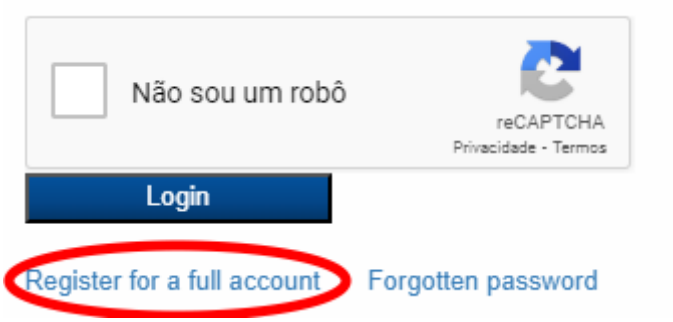
É possível ainda escolher ferramentas de simulação entre várias disponíveis, incluindo *Synopsys VCS*, *Cadence Incisive*, e *Mentor Graphics ModelSim*, que são simuladores

amplamente utilizados na indústria. Pode-se usar bibliotecas e modelos prontos para testes e verificação, bem como compartilhar projetos facilmente com colegas, professores ou em apresentações, já que ele gera um *link* público para seu código e resultados de simulação. É uma ferramenta valiosa para aprendizado e prototipagem rápida, principalmente para quem deseja aprender ou *praticar design* e verificação de circuitos digitais sem precisar configurar um ambiente local de desenvolvimento EDA.

Para realizar as simulações em VHDL do roteiro, será necessário criar uma conta na plataforma, pois o acesso aos simuladores comerciais (os gratuitos não funcionam bem com o VHDL) só é dado para portadores de contas. Além disso, a conta permite guardar os *Playgrounds* (que é o nome dado aos projetos de simulação), bem como compartilhá-los através de *links*. Abaixo está um guia passo-a-passo para criar uma conta e um novo *Playground* devidamente configurado para uma simulação VHDL.

IMPORTANTE: Preferencialmente, abra uma janela privativa no navegador ("Open a New Private Window"). Após feito o *login* na conta, parece que o *logout* não funciona muito bem. Em uma janela normal, sua conta ficará disponível para outros. Com a janela privativa, o acesso da conta é "esquecido" pelo navegador, demandando novo *login*.

1. Acesse o *site* https://www.edaplayground.com. Na frente da página principal, abrirá uma janela de boas-vindas sugerindo a criação de uma conta. **NÃO** realize o login com Google! Para poder usar as melhores ferramentas de simulação e síntese, será necessário criar uma conta com um email institucional, e o e-mail da Unicamp serve perfeitamente para esta finalidade. Desça a janela até o final. Logo abaixo do botão de *Login* há um *link* chamado "Register for a full account".

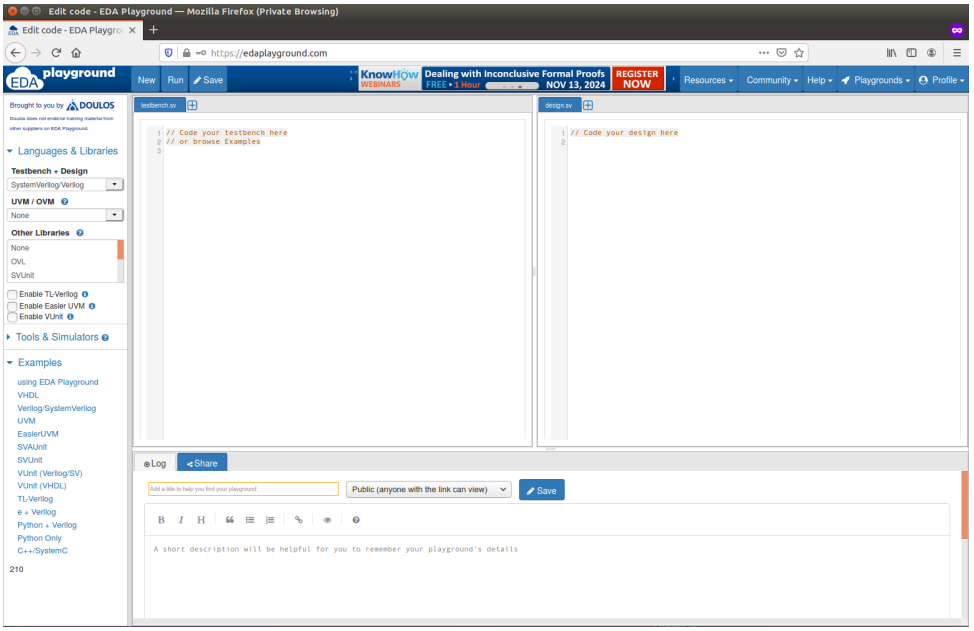


2. Na página que se abre, informe seus dados para a conta, começando com o e-mail institucional (o da DAC), e criando uma senha, além do nome da instituição ("University of Campinas"), nome, cidade e país. Se for obrigatório o *Job Title*, pode-se preencher com "Student". No final da página, selecione o *Captcha* e clique no botão "I Agree".

3. Após fornecer os dados, a página informa que o email deve ser verificado. Feche a página, vá ao seu email e clique no *link* de verificação. A página do EDA Playground abrirá

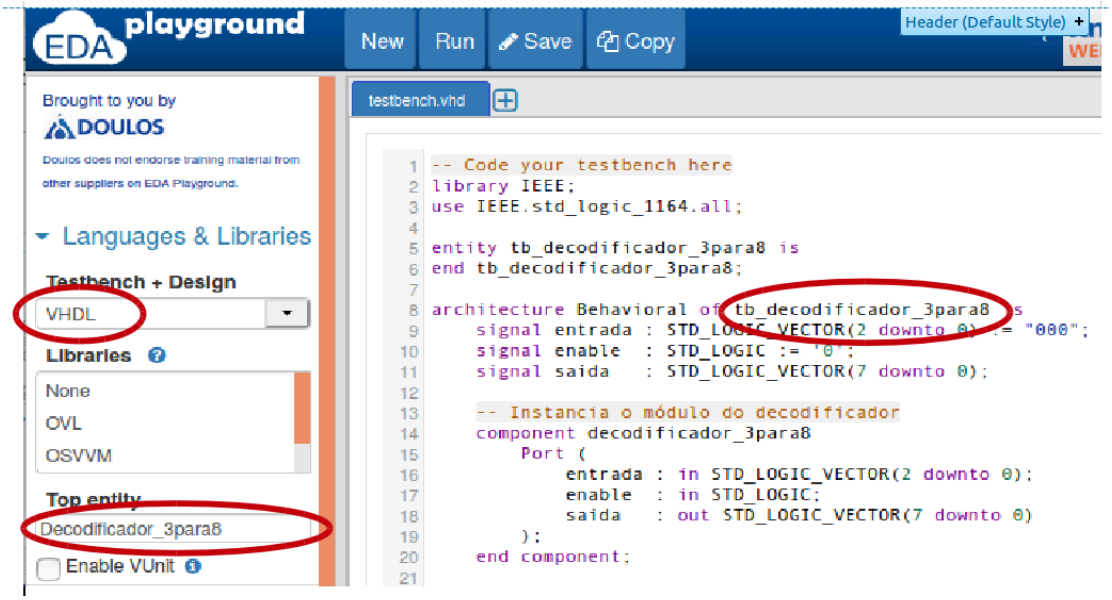
novamente, já logada em sua conta. Quando for entrar outra vez, na página inicial basta fornecer o email e a senha e depois confirmar o *Captcha*.

4. O painel principal tem em sua parte central duas janelas de edição. A janela da esquerda é para o arquivo de *testbench* (que determina as condições de simulação e fornece os sinais nas entradas do *hardware* simulado). A janela da direita é para o arquivo principal com a descrição do *hardware*. Os comentários iniciais nos textos das janelas descrevem suas finalidades.

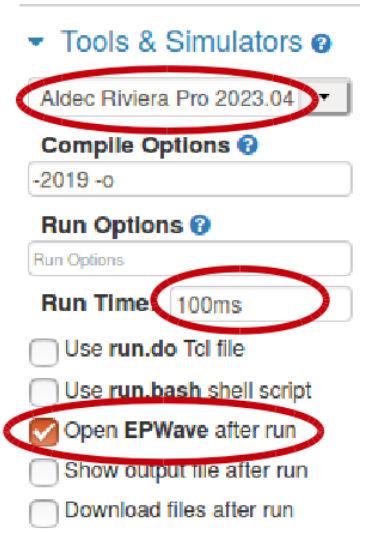


5. Na barra superior, há alguns botões. Os mais importantes são: *New* (novo projeto), *Run* (executar simulação), *Save* (guardar projeto), "Playgrounds" (acessa a lista de projetos) e "Profile" (configura o perfil). Vamos inicialmente em "Profile". Na página que se abrir, marque a caixa "Open EPWave waveforms on a separate page after run". Esta opção faz com que as formas de onda simuladas sejam abertas em outra aba no navegador, e não por cima da janela principal. Depois clique no botão "Save Details" para salvar a configuração e, na parte superior, no botão "New" para voltar à tela principal.

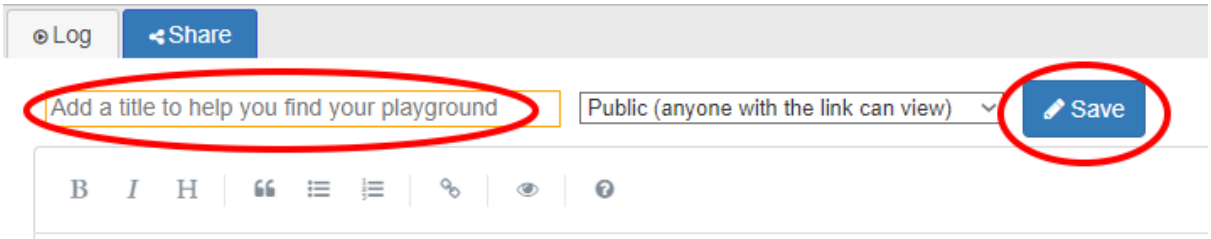
6. No painel à esquerda, estão as opções gerais para a simulação. Inicialmente, expanda o menu ***Languages & Libraries***. A opção ***Testbench + Design*** define a linguagem de descrição que será utilizada, sendo que o padrão é o Verilog. Selecione a opção "**VHDL**" e note que os textos nas janelas de edição mudam. Agora os comentários seguem o padrão VHDL e já foi incluída a biblioteca padrão. As demais opções podem permanecer inalteradas, mas é necessário adicionar informação no campo ***Top entity***. Este campo determina qual é a entidade VHDL que ocupa o topo da hierarquia. Em cada uma de nossas simulações, será criada uma entidade para o *testbench*, que será o topo da hierarquia de testes do projeto. Basta copiar o nome (que se segue à palavra chave ***entity***) e colar no campo do painel à esquerda. O nome de entidade que vamos projetar é `decodificador_3para8` e a sua arquitetura de banco de testes será denominada `decodificador_3para8_tb`.



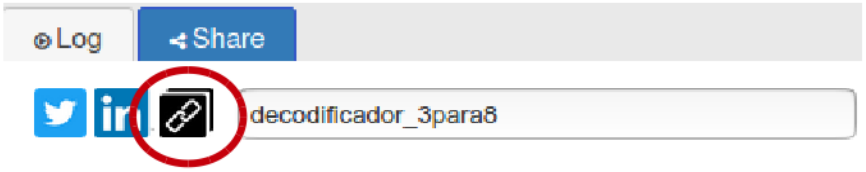
7. Agora expanda o menu ***Tools & Simulators***. O primeiro item é um *dropdown* com a palavra "Select…". Clique nele para selecionar o simulador a ser usado. Nas simulações deste roteiro, foi usado o ***Aldec Riviera Pro 2023.04***. Mantenha as demais opções no padrão, exceto a ***Run Time***, que deve ser ajustada para o tempo total de simulação, e a caixa ***Open EPWave after run***, que indica que após a simulação deve-se abrir a aba com as formas de onda.



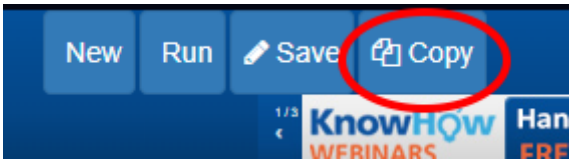
8. Com as configurações ajustadas, pode-se gravar o *Playground*. No painel central inferior, na aba "Share", há um campo para escrever o nome do *Playground*. Escreva "decodificador_3para8", clicando-se logo depois no botão "Save".

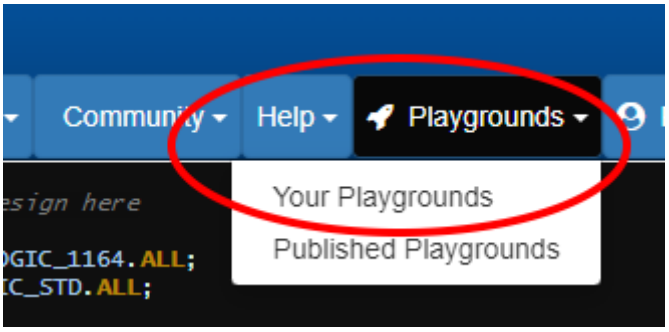


Após isso, aparecem no mesmo painel ícones de redes sociais e um ícone com uma corrente, para compartilhar o *Playground*. O ícone da corrente copia o *link* da página do *Playground* para a área de transferência.



Qualquer pessoa, estando logada em sua própria conta, pode abrir o *link* e clicar no botão "Copy" do menu superior para guardar uma cópia do *Playground* original em sua própria conta. Todos os *Playgrounds* gravados podem ser acessados através do botão "Playgrounds", opção "Your Playgrounds".





9. Neste ponto, deve-se criar o código do VHDL (*design.vhd*) e do *testbench*. Na janela da **direita** coloque o primeiro código, que descreve um decodificador de 3 para 8 linhas com *enable*. Neste decodificador, todas as linhas permanecem em nível alto exceto aquela cujo número corresponde ao valor binário das 3 entradas. Caso o *enable* tenha o valor baixo, todas as saídas permanecem em nível alto.

```
-- Code your design here
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity decodificador_3para8 is
```

```
    Port (
        entrada : in STD_LOGIC_VECTOR(2 downto 0);  -- Entrada de 3 bits
        enable  : in STD_LOGIC;                     -- Sinal de habilitação
        saida   : out STD_LOGIC_VECTOR(7 downto 0)  -- Saída de 8 bits
    );
end decodificador_3para8;

architecture Behavioral of decodificador_3para8 is
begin
    process(entrada, enable)
    begin
        if enable = '1' then
            case entrada is
                when "000" => saida <= "11111110";
                when "001" => saida <= "11111101";
                when "010" => saida <= "11111011";
                when "011" => saida <= "11110111";
                when "100" => saida <= "11101111";
                when "101" => saida <= "11011111";
                when "110" => saida <= "10111111";
                when "111" => saida <= "01111111";
                when others => saida <= "11111111";
            end case;
        else
            saida <= "11111111";  -- Saída 1 quando enable é 0
        end if;
    end process;
end Behavioral;
```

Na janela da **esquerda** coloque o código do *testbench*. Neste código é criada uma sequência de valores nas entradas digitais para cobrir todas as possibilidades. A duração de cada grupo de valores é de 15ms.

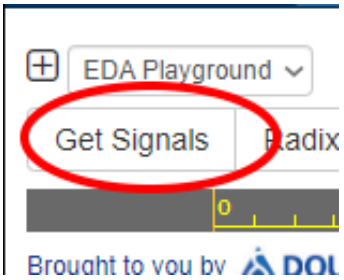
```
-- Code your testbench here
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity tb_decodificador_3para8 is
end tb_decodificador_3para8;

architecture Behavioral of tb_decodificador_3para8 is
    signal entrada : STD_LOGIC_VECTOR(2 downto 0) := "000";
    signal enable  : STD_LOGIC := '0';
    signal saida   : STD_LOGIC_VECTOR(7 downto 0);
```

```
    -- Instancia o módulo do decodificador
    component decodificador_3para8
        Port (
            entrada : in STD_LOGIC_VECTOR(2 downto 0);
            enable  : in STD_LOGIC;
            saida   : out STD_LOGIC_VECTOR(7 downto 0)
        );
    end component;

begin
    uut: decodificador_3para8 Port map (
        entrada => entrada,
        enable => enable,
        saida => saida
    );

    -- Estímulo de teste
    process
    begin
        -- Ativa o enable e verifica todas as entradas
        enable <= '1';

        wait for 15 ms; entrada <= "000";
        wait for 15 ms; entrada <= "001";
        wait for 15 ms; entrada <= "010";
        wait for 15 ms; entrada <= "011";
        wait for 15 ms; entrada <= "100";
        wait for 15 ms; entrada <= "101";
        wait for 15 ms; entrada <= "110";
        wait for 15 ms; entrada <= "111";

        -- Desativa o enable e observa a saída zerada
        wait for 15 ms; enable <= '0';
        entrada <= "000";

        -- Finaliza a simulação
        wait for 15 ms;
        wait;
    end process;
end Behavioral;
```

São 10 possibilidades testadas, portanto o tempo de simulação deve ser ajustado em 100ms. O nome da “Top entity” deve ser “tb_decodificador_3para8”. Depois, pode-se salvar tudo,

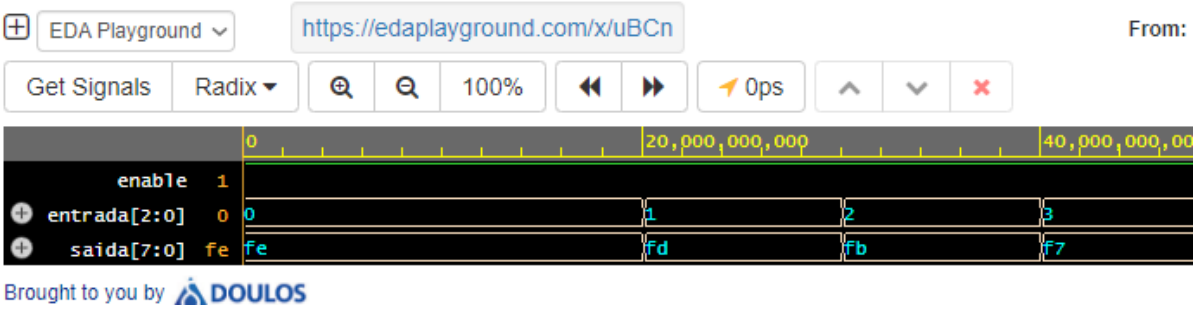
clicando no botão “Save”, e usar o botão “Run” no lado esquerdo da aba superior para simular o projeto. Após concluída a simulação (caso não haja erros), abre-se nova aba com as formas de onda.
Obs: Caso use um navegador com bloqueador de *pop-ups*, será necessário permitir os *pop-ups* para o site www.edaplayground.com.

10. Na janela das formas de onda, os sinais podem aparecer ou não. Caso não apareçam os sinais, será necessário selecionar os sinais a serem apresentados. Clique no botão “Get Signals” e na janela que se abre, à esquerda (***Scope***) selecione o nome da entidade do *testbench*. Os nomes dos sinais do *testbench* vão aparecer. Clique no botão “Append All”, para incluir todos os sinais.



Há ainda um segundo nome no ***Scope***: .UUT (do inglês *Unit Under Test*). O conjunto de sinais da entidade do *testbench* se refere às portas de entrada e saída externas do *design* de nível superior, permitindo que monitoremos como o circuito interage com o ambiente simulado em um nível macro. Já o conjunto .DUT oferece acesso a **todos os sinais internos e externos** da instância do projeto que está sendo testada. Essa capacidade de visualizar os sinais internos do .UUT, ou .DUT (do inglês *Device Under Test*), é fundamental para **identificar e diagnosticar problemas** que não são visíveis apenas pelas portas externas, permitindo uma análise granular e a validação detalhada do comportamento de cada componente interno do seu projeto. A relevância reside na habilidade de alternar entre uma visão de alto nível do sistema e uma inspeção minuciosa dos seus componentes, otimizando o processo de verificação do *hardware*.

Depois use as ferramentas de *zoom* (lupas) e cursores de tempo (CTRL-Click).



Os sinais podem ser apagados (selecionar a linha do sinal e clina em ‘x’ laranja) e podem ter sua ordem modificada. Os agrupamentos de sinais podem ser expandidos para os sinais individuais e vice-versa. A interface é bastante intuitiva.

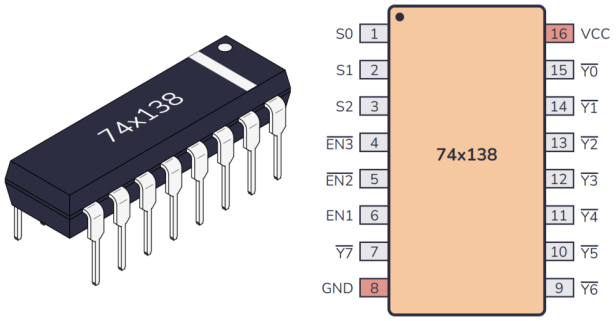
11. As formas de onda exibidas na imagem do item 10 não mostram todas as 10 possibilidades de saída. Quais (re)configurações seriam necessárias para exibir todas as saídas possíveis?

12. Se tentarmos interpretar a descrição em HDL do item 9 sob a ótica de um programador acostumado a microcontroladores, poderíamos supor, equivocadamente, que o sinal de saída será atualizado apenas após alguns ciclos de relógio, talvez dois ou três, após a "execução" de uma instrução que compara o valor da variável `enable` com '1'.

```
architecture Behavioral of decodificador_3para8 is
begin
    process(entrada, enable)
    begin
        if enable = '1' then
            case entrada is
                when "000" => saida <= "11111110";
                when "001" => saida <= "11111101";
                when "010" => saida <= "11111011";
                when "011" => saida <= "11110111";
                when "100" => saida <= "11101111";
                when "101" => saida <= "11011111";
                when "110" => saida <= "10111111";
                when "111" => saida <= "01111111";
                when others => saida <= "11111111";
            end case;
        else
            saida <= "11111111";  -- Saída 1 quando enable é 0
        end if;
    end process;
end Behavioral;
```

Esse raciocínio faz sentido em linguagens imperativas como C, onde há uma sequência clara de execução passo a passo. No entanto, no contexto de uma HDL, essa leitura deve ser completamente diferente. Aqui, não existe uma linha de execução sequencial tradicional. O código HDL descreve um comportamento estrutural e paralelo que será fisicamente sintetizado em lógica digital. Em vez de "executar" instruções, estamos descrevendo como o circuito será construído — e esse circuito reage continuamente às mudanças nas entradas, propagando sinais de acordo com as conexões definidas.

Para entender melhor, imagine que essa descrição seja sintetizada em um circuito combinacional semelhante a um decodificador 74138. Nesse cenário, a lógica é implementada fisicamente com portas lógicas conectadas de forma fixa. O que acontece com o valor de saída tão logo os sinais de entrada mudam? São necessários ciclos de execução? Como os resultados se propagam pelas portas lógicas até atingir a saída? Com esse novo olhar, observe atentamente as formas de onda geradas pelo simulador. Quando a saída 0b11111110 é ativada ao colocarmos 0b000 na entrada? Isso reflete o comportamento típico de um circuito combinacional: as saídas respondem automaticamente às entradas, dentro dos limites físicos do tempo de propagação.



Fonte: buildelectronicircuits

Portanto, para interpretar corretamente sistemas descritos em HDL, é essencial abandonar o modelo mental sequencial de execução e adotar uma perspectiva de ativação simultânea e reatividade contínua. Isso é o que diferencia fundamentalmente uma linguagem de descrição de *hardware* de uma linguagem de programação para microcontroladores.

13. No *testbench* de especificação de testes, modifique o intervalo dos sinais de 15ms para 10ms. Em seguida, execute a simulação ("Run") e compare as formas de onda geradas neste teste com as do teste anterior. Como os valores dos parâmetros de teste podem afetar nos resultados de simulação? Como os valores dos parâmetros de teste podem afetar os resultados da simulação?

Note que os valores dos parâmetros definidos no *testbench* controlam os **estímulos de entrada** que o dispositivo sob teste (DUT) ou a unidade sob teste (UUT) recebe.

14. No arquivo de especificação do circuito, `design.vhd`, modifique a saída nas linhas 23 e 24 conforme indicado na seguinte figura. Em seguida, execute a síntese e a simulação ("Run") e compare as formas de onda geradas neste teste com as do teste anterior. Como a descrição do comportamento de um circuito afeta os resultados da simulação?

A descrição do comportamento de um circuito em uma linguagem de descrição de *hardware* (HDL), como VHDL, é a definição literal de como seu *hardware* funcionará. Qualquer alteração nessa descrição altera diretamente a lógica que o simulador (e posteriormente o *hardware* físico) irá implementar.

```
18          case entrada is
19              when "000" => saida <= "11111110";
20              when "001" => saida <= "11111101";
21              when "010" => saida <= "11111011";
22              when "011" => saida <= "11110111";
23              when "100" => saida <= "11100011";
24              when "101" => saida <= "11011001";
25              when "110" => saida <= "10111111";
26              when "111" => saida <= "01111111";
27              when others => saida <= "11111111";
```

15. Ao compilar os arquivos VHDL, o EDA *Playground* verifica a sintaxe do código e gera mensagens de erro que auxiliam na localização e correção de falhas. Apague ";" no final da linha 27 do `design.vhd` na janela direita

```
25                when "110" => saida <= "10111111";
26                when "111" => saida <= "01111111";
27                when others => saida <= "11111111"
28            end case;
29        else
```

Execute a síntese do código ("Run") e observe as mensagens geradas na aba "Log". Você consegue identificar a fonte do erro através do *log* do sintetizador?

```
Log   Share
work = /home/runner/work/work.lib
Aldec, Inc. VHDL Compiler, build 2023.04.112
VLM Initialized with path: "/home/runner/library.cfg".
COMP96 Compile Entity "decodificador_3para8"
COMP96 Compile Architecture "Behavioral" of Entity "decodificador_3para8"
COMP96 Compile Entity "tb_decodificador_3para8"
COMP96 Compile Architecture "Behavioral" of Entity "tb_decodificador_3para8"
COMP96 Top-level unit(s) detected:
COMP96 Entity => tb_decodificador_3para8
COMP96 ERROR COMP96_0015: "';' expected." "design.vhd" 28 13
COMP96 Compile failure 1 Errors 0 Warnings  Analysis time :  20.0 [ms]
Exit code expected: 0, received: 255
```

Projeto 2: Descrevendo e simulando um circuito sequencial

Diferentemente das linguagens de programação tradicionais, onde a execução segue uma ordem sequencial de comandos, em uma linguagem de descrição de *hardware*, as instruções não são executadas em sequência, mas representam estruturas que operam simultaneamente. No projeto `decodificador_3para8`, cada declaração no código VHDL descreve um componente ou comportamento que será sintetizado como parte de um sistema de *hardware* operando em paralelo. Assim, a síntese do sistema a partir do código VHDL gera uma arquitetura composta por módulos paralelos e interconectados, não uma linha de execução sequencial.

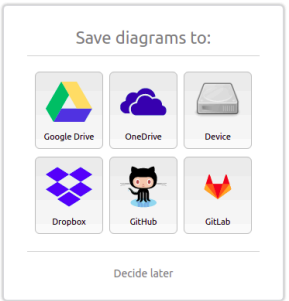
Essa forma de pensar é essencial para compreender o **circuito combinacional**, onde as saídas dependem exclusivamente do estado atual das entradas. No entanto, o mundo digital que nos cerca, como processadores, controladores, memórias e interfaces, vai muito além disso. Esses sistemas não apenas reagem a sinais de entrada, mas evoluem ao longo do tempo, armazenam estados e tomam decisões com base em eventos passados. Para isso, eles dependem de uma estrutura fundamental: o circuito sequencial. Diante dessa transição do comportamento puramente combinacional para o comportamento com memória e evolução temporal, surge um novo desafio: como usar HDL para descrever e sintetizar circuitos sequenciais, que respondem não só às entradas atuais, mas também ao histórico de sinais, controlados por um sinal de relógio (*clock*)? A partir do que aprendemos sobre circuitos combinacionais,

projetamos agora, em VHDL, um circuito sequencial simples: um **detector de sequência de 6 *bits***, implementado como uma máquina de estados finitos (FSM).

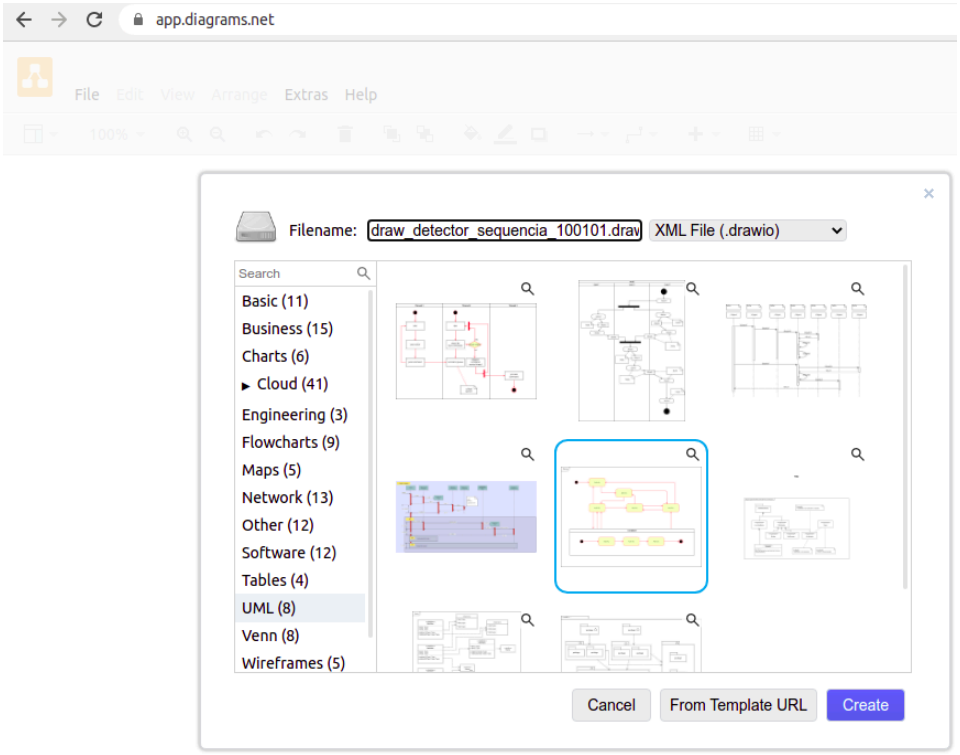
Para lidar com essa nova complexidade de forma estruturada e compreensível, podemos recorrer a uma abordagem complementar de modelagem. A combinação de UML (do inglês *Unified Modeling Language*) e VHDL nos permite modelar e implementar máquinas de estados em circuitos físicos reais. Por meio da UML, criamos diagramas de máquina de estados que representam de forma concisa e intuitiva os diferentes estados do sistema e suas transições, facilitando a compreensão e a comunicação entre os membros da equipe de projeto. Uma vez que a máquina de estados está claramente definida, o VHDL entra em cena para descrever a lógica do circuito de maneira estruturada e sintetizável. O próprio VHDL oferece recursos nativos para a implementação de máquinas de estados, como a possibilidade de descrever estados, transições e comportamentos associados em processos sensíveis ao *clock*, o que facilita a tradução direta de modelos abstratos em circuitos digitais confiáveis. Assim, conseguimos criar uma descrição precisa do sistema, viabilizando sua construção física sem a necessidade de lidar diretamente com os detalhes elétricos de baixo nível.

O aplicativo draw.io, agora conhecido como diagrams.net, é uma ferramenta *online* gratuita para criação de diagramas e gráficos. Ela oferece um editor gráfico intuitivo que permite a criação de uma ampla gama de diagramas, como fluxogramas, diagramas de rede, organogramas, diagramas de processos, além de ser frequentemente usada para criar diagramas da UML. Embora o draw.io ofereça suporte para muitos tipos de diagramas UML, sua compatibilidade não é 100%, o que significa que alguns elementos e padrões da UML podem não ser totalmente precisos ou disponíveis. O aplicativo é baseado em navegador e também permite o salvamento e compartilhamento de diagramas em várias plataformas, como *Google Drive*, *OneDrive* e localmente.

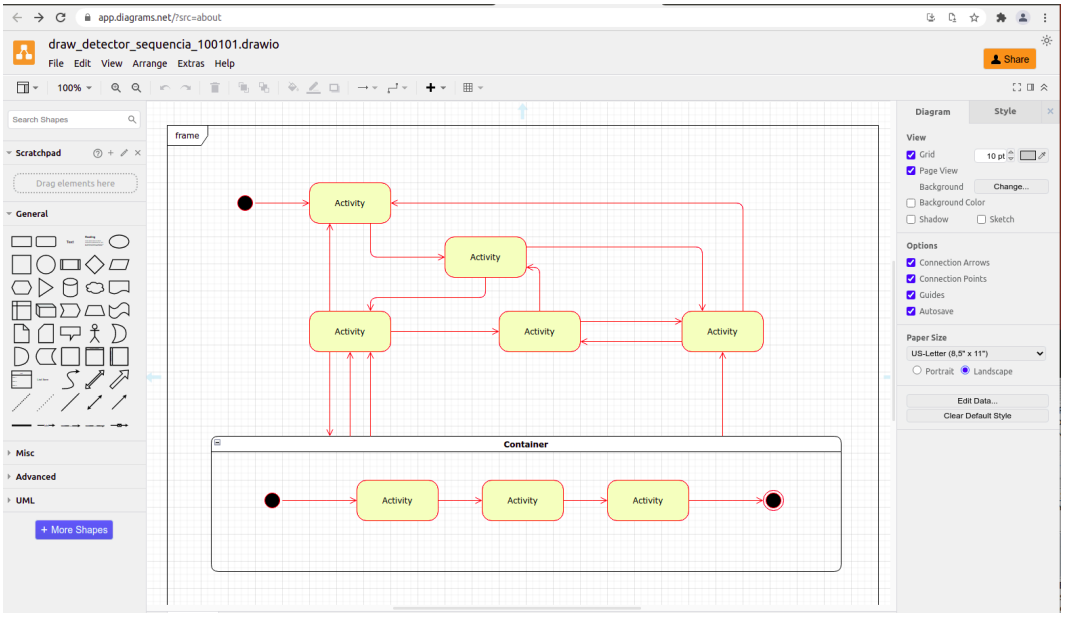
Para ilustrar a aplicação combinada de UML e VHDL no desenvolvimento de um circuito sequencial, vamos representar primeiro uma **máquina de estados** correspondente ao **detector de sequência de 6 *bits*** "100101". Utilizamos o draw.io, acessando o link para iniciar seu diagrama e escolher onde salvá-lo.



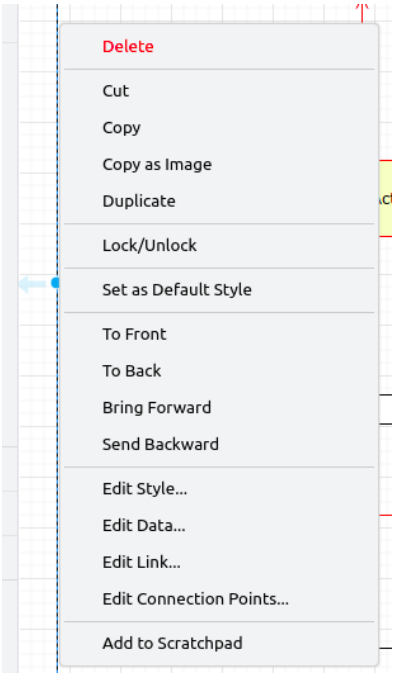
Ao escolher um dispositivo de armazenamento, novos menus *popup* aparecerão para você atribuir um nome ao projeto (por exemplo, `draw_detector_sequencia_100101.drawio`), especificar o local de armazenamento, criar um arquivo de armazenamento no local selecionado (por exemplo, `draw_detector_sequencia_100101.drawio`), e se quiser, escolher um *template* para o seu projeto. Neste caso, foi selecionado um *template* de diagrama de máquina de estados da UML para reaproveitar os componentes já incluídos.



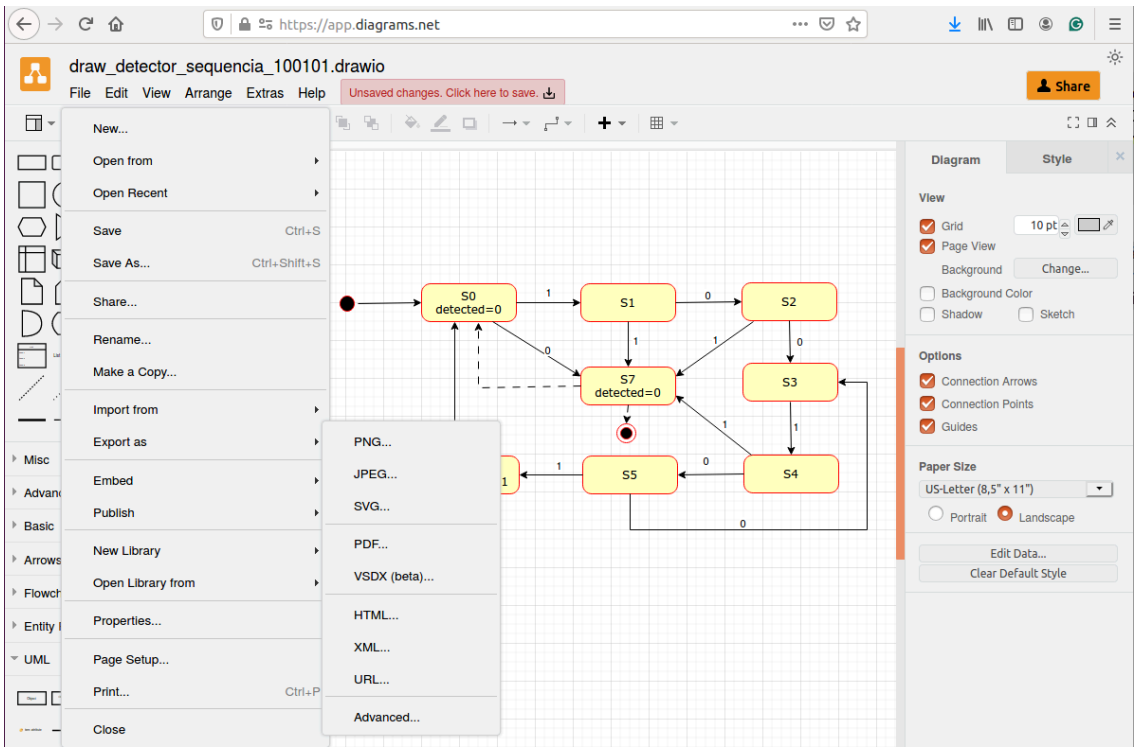
Abre-se então uma nova tela com o *template* selecionado no meio da tela. No centro da interface está o editor gráfico, onde o diagrama é desenhado e editado. À esquerda, há uma barra de ferramentas que contém uma variedade de elementos gráficos que podem ser arrastados para o editor. Esses elementos incluem formas básicas (como retângulos, círculos, etc.), conexões, ícones e outros objetos que podem ser inseridos no diagrama com um simples clique ou arraste. À direita, a aba permite configurar a aparência e o estilo dos elementos gráficos selecionados. É possível ajustar propriedades como cor, tamanho, fonte, borda e outros aspectos visuais de cada objeto inserido, além de realizar configurações mais avançadas.



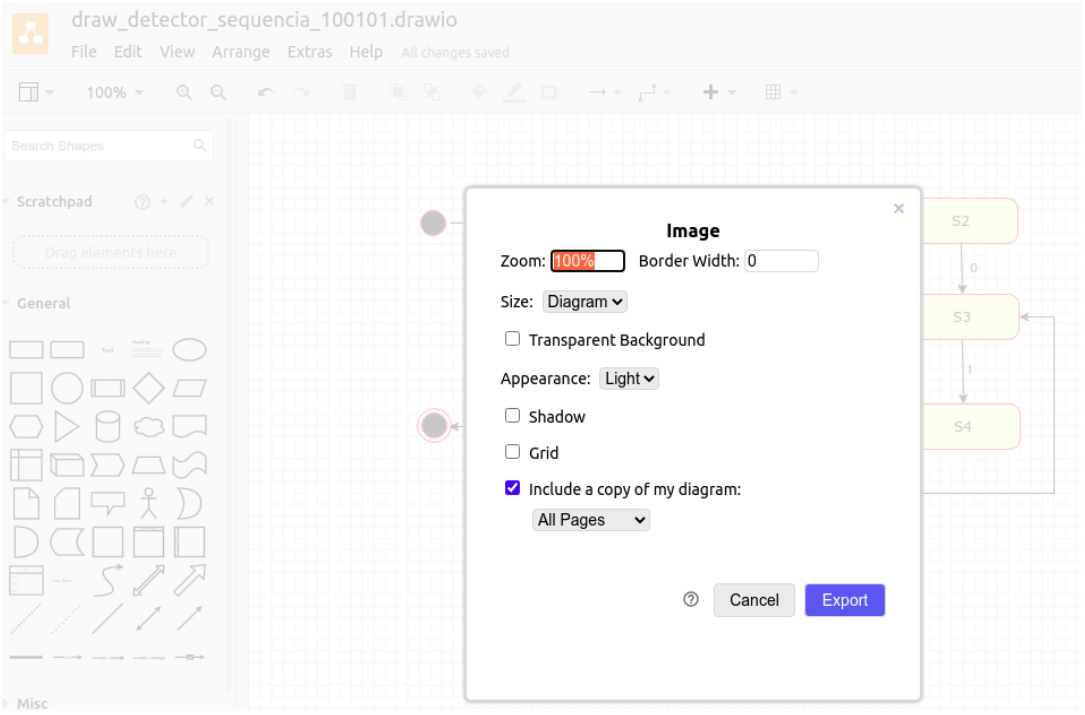
Observe que o diagrama está "embutido" em um `frame`. Para editá-lo, clique uma vez sobre o `frame` e, em seguida, pressione o botão direito do *mouse* para abrir o menu de contexto. Selecione a opção "Delete" para liberar os elementos gráficos e permitir a edição.



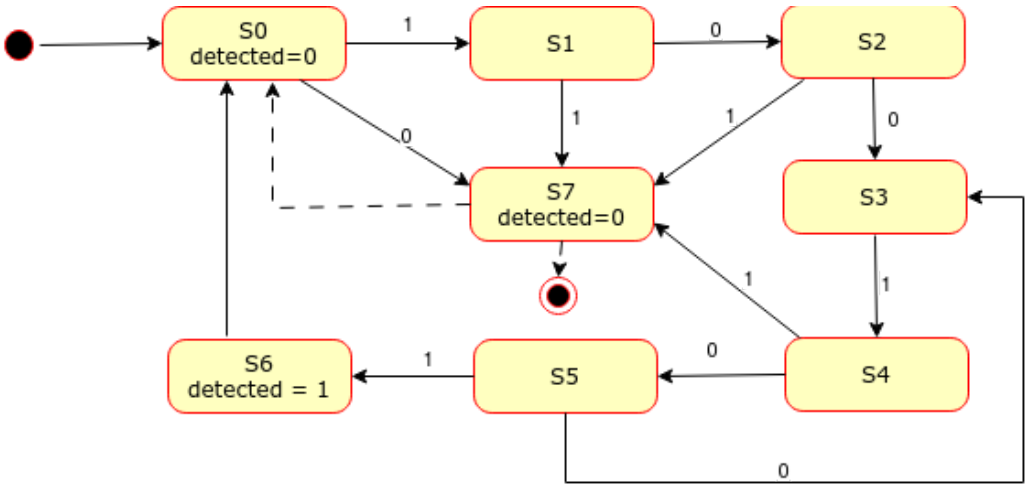
Utilizando os recursos disponíveis no editor, foi criado o diagrama de máquina de estados do projeto. Para salvar o arquivo, abra o menu *dropdown* "File" na barra superior. Ao selecionar a opção "Save As…", o sistema permitirá que você escolha o local para salvar o arquivo. Já ao escolher "Export as…", serão apresentadas diferentes opções de formatos para exportação.



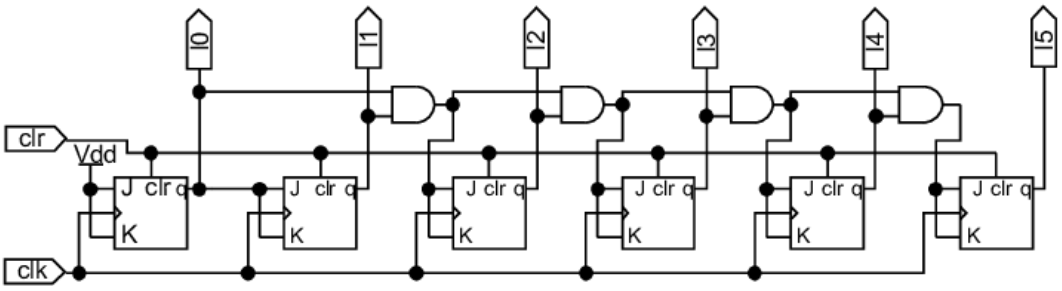
Em seguida, é apresentado um menu de opções para a imagem a ser salva.



Segue abaixo o diagrama de máquina de estados implementado no projeto `detector_sequencia_100101`, que apresenta duas transições indefinidas para o estado S7. As linhas pontilhadas indicam as possíveis alternativas de transição que o sistema pode seguir ao estar no estado S7.



Vamos transformar o nosso diagrama de máquina de estados em um circuito sequencial físico de detecção de sequência de *bits*. Para isso, a maneira mais clara e direta é através de um esquemático gráfico, que ilustra como o diagrama de estados modelado pode ser realizado em *hardware*. O diagrama a seguir ilustra a implementação desse detector, utilizando um total de 12 componentes discretos: seis *flip-flops* JK e seis portas AND. A cada pulso de *clock*, um novo *bit* de entrada é inserido no primeiro *flip-flop*, enquanto os *bits* anteriores se deslocam para os *flip-flops* subsequentes. Dessa forma, os seis *flip-flops* armazenam continuamente os últimos seis *bits* recebidos. Um bloco de lógica combinacional, então, compara o conteúdo desses *flip-flops* com a sequência desejada (`"100101"`) e ativa uma saída (`detect` = '1') quando a correspondência é encontrada.



Fonte: ResearchGate

Alternativamente, podemos sintetizar o mesmo diagrama de estados em um PLD, como uma FPGA, utilizando um Ambiente de Desenvolvimento Integrado (em inglês, *Integrated Development Environment*) como o Intel Quartus Prime. Esses ambientes são projetados para projetar, simular, sintetizar e implementar circuitos lógicos em PLDs. O Quartus Prime suporta tanto diagramas esquemáticos quanto as linguagens de descrição de *hardware* mais populares, Verilog e VHDL. Isso significa que ele é capaz de traduzir tanto representações gráficas quanto descrições textuais em circuitos lógicos prontos para serem implementados em um PLD. Por exemplo, o esquemático do detector que acabamos de ver é suficiente para que o IDE gere o circuito necessário na FPGA.

Agora, vamos explorar a descrição textual desse detector sem o sinal `clr`, mas utilizando a linguagem VHDL, que também pode ser implementada em uma FPGA através de um IDE.

1. Crie um novo projeto no EDA Playground usando o botão “New”. Configure o projeto como no projeto anterior, atribuindo à “Top entity” o nome `tb__detector_sequencia_100101`.

2. Na janela à direita, escreva a descrição do detector da sequência de *bits* "100101", que é uma tradução direta e textual do que está representado graficamente no diagrama de estados.

```
-- Code your design here

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity detector_sequencia_100101 is
    Port (
        clk      : in  STD_LOGIC;  -- Sinal de clock
        bit_in   : in  STD_LOGIC;  -- Entrada do bit a ser verificado
         detected : out STD_LOGIC   -- Saída que vai para 1 durante um ciclo de clock quando a
sequência "100101" é detectada
    );
end detector_sequencia_100101;

architecture Behavioral of detector_sequencia_100101 is
    -- Definindo os estados do detector
    type state_type is (S0, S1, S2, S3, S4, S5, S6);
    signal state : state_type := S0;
begin
    process(clk)
    begin
        if rising_edge(clk) then
            case state is
                when S0 =>
                    if bit_in = '1' then
                        state <= S1;
                    end if;
                    detected <= '0';
                when S1 =>
                    if bit_in = '0' then
                        state <= S2;
                    else
                        state <= S1;
                    end if;
```

```
                detected <= '0';
            when S2 =>
                if bit_in = '0' then
                    state <= S3;
                else
                    state <= S1;
                end if;
                detected <= '0';
            when S3 =>
                if bit_in = '1' then
                    state <= S4;
                else
                    state <= S0;
                end if;
                detected <= '0';
            when S4 =>
                if bit_in = '0' then
                    state <= S5;
                else
                    state <= S1;
                end if;
                detected <= '0';
            when S5 =>
                if bit_in = '1' then
                    state <= S6;
                    detected <= '1';  -- Sequência "100101" detectada, saída em 1 por um ciclo
                else
                    state <= S3;
                    detected <= '0';
                end if;
            when S6 =>
                detected <= '0';  -- Reseta a saída após um ciclo
                state <= S0;      -- Reinicia a sequência
        end case;
    end if;
  end process;
end Behavioral;
```

Essencialmente, cada estado (S_i) do diagrama corresponde a um *flip-flop* JK na nossa implementação com componentes discretos. A lógica AND, realizada pelas portas lógicas, é descrita usando a declaração sequencial `if ... then ... end if` em VHDL. As transições entre os estados são coordenadas pela borda de subida do sinal de relógio (`clk`), o que é expresso pela condicional `if rising_edge(clk) then ... end if`. É essa

lógica sequencial sensível à borda do *clock* que é responsável pela inferência de *flip-flops* na síntese do circuito pelo sintetizador VHDL. Dentro da declaração sequencial `case`, cada transição de estado do *flip-flop* é detalhada em um ramo específico `when`, que, por sua vez, contém o bloco combinacional da porta lógica correspondente.

3. Na janela à **esquerda**, adicione um código da *testbench*:

```
-- Code your testbench here
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity tb_detector_sequencia_100101 is
end tb_detector_sequencia_100101;

architecture Behavioral of tb_detector_sequencia_100101 is
    signal clk      : STD_LOGIC := '0';
    signal bit_in   : STD_LOGIC := '0';
    signal detected : STD_LOGIC;

    -- Instancia o módulo do detector
    component detector_sequencia_100101
        Port (
            clk      : in  STD_LOGIC;
            bit_in   : in  STD_LOGIC;
            detected : out STD_LOGIC
        );
    end component;

begin
    uut: detector_sequencia_100101 Port map (
        clk => clk,
        bit_in => bit_in,
        detected => detected
    );

    -- Gera o clock de 10ns
    clk_process : process
    begin
        clk <= '0';
        wait for 5 ns;
        clk <= '1';
        wait for 5 ns;
    end process;

    -- Estímulo de teste
```

```
    process
    begin
        -- Aplica a sequência "100101" com outros bits entre as tentativas
        wait for 10 ns; bit_in <= '1';
        wait for 10 ns; bit_in <= '0';
        wait for 10 ns; bit_in <= '0';
        wait for 10 ns; bit_in <= '1';
        wait for 10 ns; bit_in <= '0';
          wait for 10 ns; bit_in <= '1'; -- Sequência "100101" detectada, 'detected' vai a 1 por um
ciclo

        wait for 10 ns; bit_in <= '0'; -- Reinicia a sequência
        wait for 10 ns; bit_in <= '1';
        wait for 10 ns; bit_in <= '0';
        wait for 10 ns; bit_in <= '0';
        wait for 10 ns; bit_in <= '1';
        wait for 10 ns; bit_in <= '0';
        wait for 10 ns; bit_in <= '1'; -- Sequência "100101" detectada novamente, 'detected' vai a
1 por um ciclo

        -- Finaliza a simulação
        wait for 1000 ns;
        wait;
    end process;
end Behavioral;
```

4. Ajuste o tempo de execução para 200ns e renomeie a “Top entity” para `tb_detector_sequencia_100101`. Em seguida, execute a simulação, analise as formas de onda e responda:

- Os sinais de saída reagem imediatamente às entradas, desconsiderando os atrasos de propagação?
- Em quais instantes o sinal de saída muda de estado?
- Como o comportamento do circuito está vinculado ao sinal de clock?
- É possível observar como os estados internos se modificam ao longo do tempo?

Agora, compare esse comportamento com o circuito combinacional `decodificador_3para8`. No circuito sequencial, o tempo, marcado pelo `clk`, tem um papel essencial na evolução do sistema. Isso nos leva a uma reflexão: o tempo pode afetar a lógica de um circuito? Em outras palavras, será que um mesmo circuito pode apresentar um comportamento logicamente diferente quando consideramos apenas relações puras entre sinais, em contraste com quando introduzimos a dimensão temporal e estados internos?

5. Os elementos em uma descrição VHDL que caracterizam um circuito sequencial, em contraste com um combinacional, são aqueles que implicam a existência de memória (estado) e sincronismo (*clock*). Ao comparar a descrição do projeto combinacional `decodificador_3para8` com a do `detector_sequencia_100101`, notamos que, embora ambos empreguem o bloco `process` com uma lista de sinais de sensibilidade, **três elementos adicionais** são cruciais para caracterizar o `detector_sequencia_100101` como um circuito sequencial. Quais são esses três elementos, excluindo a criação de um tipo personalizado para os identificadores de estado?

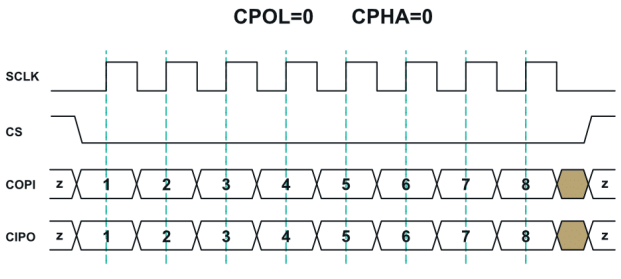
6. Quais alterações você faria na descrição `detector_sequencia_100101.vhd` para que a sequência seja reiniciada no estado S0 sempre que um dígito que não faça parte da sequência for detectado?

7. Qual valor você configuraria no campo de "`Run Time`" do EDA *Playground* para visualizar a simulação completa das duas sequências de *bits*?

Projeto 3: Criando um controlador na arquitetura FSMD

No Roteiro 10, exploramos a configuração e operação do módulo SPI integrado ao microcontrolador STM32H7A3. Agora, imagine como seria projetar um módulo SPI semelhante em uma FPGA, modelando-o em nível de transferência de registradores (em inglês, *Register Transfer Level*) ou em nível de máquina de estados finitos com caminho de dados (em inglês, F*inite State Machine with Datapath* – FSMD), usando a linguagem VHDL. Desafie-se a dar esse próximo passo: como criar um módulo SPI capaz de converter dados paralelos do núcleo em sinais seriais síncronos, conforme o protocolo SPI? Para isso, vamos aproveitar os recursos da FPGA e a flexibilidade da linguagem VHDL, que permite descrever o sistema em diferentes níveis, adaptando a complexidade do modelo às necessidades do projeto.

Sem perda de generalidade, vamos assumir que o nosso controlador SPI será apenas *master* e irá operar apenas no modo 0 (CPOL = CPHA = 0), transmitindo o *bit* mais significativo primeiro. O sinal \CS não é gerado pelo controlador, ficando a cargo de uma GPIO do microcontrolador. A taxa de transmissão de dados será dada pelo *clock* fornecido ao controlador (o prescaler não será implementado aqui), sendo que a quantidade de *bits* por segundo será metade da frequência do *clock*.



Note que com algumas modificações este controlador pode ter mais flexibilidade, por exemplo operando em outros modos. A taxa de transmissão depende apenas de um *prescaler* facilmente implementado. Este controlador pode ser facilmente incorporado a um microcontrolador, e geralmente é dessa forma que se projetam os módulos internos de microcontroladores, como o que foi usado nesta disciplina.

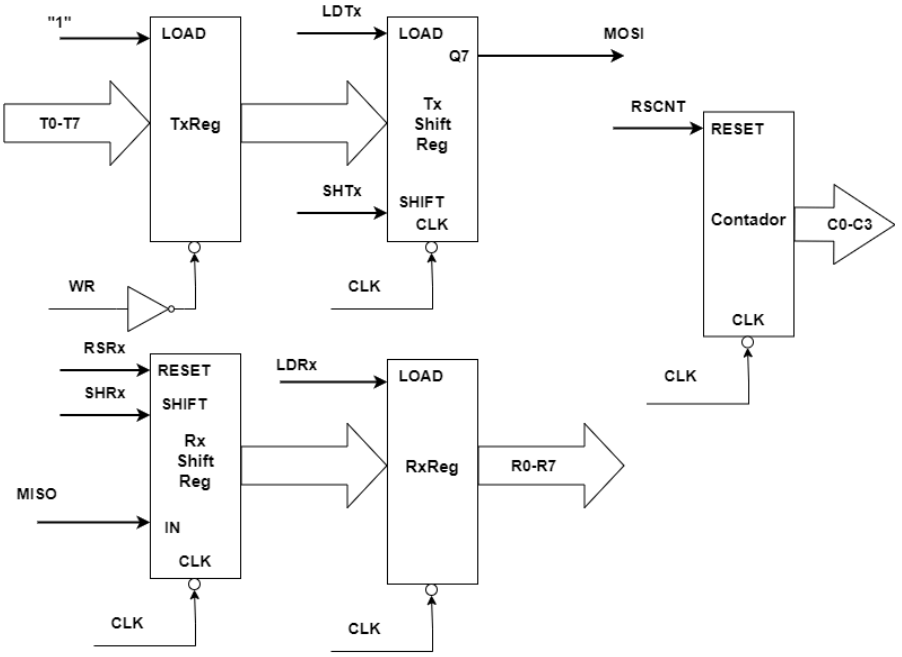
Neste projeto MasterSPI, vocês serão desafiados a analisar os resultados gerados pela solução proposta e validá-los com os resultados esperados. O foco inicial será entender os resultados práticos da implementação, ajudando a desenvolver uma visão crítica sobre o funcionamento do sistema. No entanto, é essencial que complementem essa análise com a leitura detalhada da última seção "Projeto de um Processador Dedicado". Essa seção fornece o embasamento teórico e a fundamentação técnica que explicam como o projeto foi desenvolvido até a etapa de descrição do circuito em VHDL. Compreender esse desenvolvimento é importante para entender os princípios que guiaram a criação da solução, e assim, proporcionar uma visão mais profunda sobre a construção de sistemas digitais em FPGA.

1. Inicialmente, vamos definir os sinais externos ao controlador SPI. Temos uma interface com o microcontrolador e a interface no padrão SPI. Pelo lado da interface com o microcontrolador, o controlador SPI terá uma entrada de dados de 8 *bits* para transmissão TxD (**T7-T0**) e uma saída de dados de 8 *bits* para o dado recebido RxD (**R7-R0**). Além disso, terá um sinal de entrada ***WR*** (*WRite*), cuja borda de subida inicia a transmissão do dado, e um sinal de saída ***TCPLT*** (*Transfer Complete*), gerado pela FSM, que vai para 1 quando a transmissão / recepção de *byte* acabou, indo para 0 no início da nova transferência. Há ainda uma entrada *clock* que determina o sincronismo dos elementos. Pelo lado da interface SPI, temos os sinais de dados **MOSI** e **MISO**, ligados diretamente ao *datapath*, e o sinal **SCK**, gerado pela FSM.

```
226 -- ENTIDADE SUPERIOR QUE ENGLOBA TODAS AS OUTRAS
227 library IEEE;
228 use IEEE.STD_LOGIC_1164.ALL;
229
230 entity MasterSPI is
231     Port (
232         clk : in STD_LOGIC;
233         WR : in STD_LOGIC;
234         MISO : in STD_LOGIC;
235         TxD : in STD_LOGIC_VECTOR (7 downto 0);
236         RxD : out STD_LOGIC_VECTOR (7 downto 0);
237         TCPLT : out STD_LOGIC;
238         MOSI : out STD_LOGIC;
239         SCK : out STD_LOGIC
240     );
241 end entity MasterSPI;
```

2. A entidade principal MasterSPI interconecta cinco entidades menores: `TxReg`, `TxShiftReg`, `RxShiftReg`, `RxReg` e `Contador`. Essa conexão revela o fluxo de dados entre elas, por meio de uma descrição estrutural de blocos que contêm registradores e lógica combinacional. Essa abordagem é uma característica essencial do nível de abstração RTL, onde o **caminho de dados** (em inglês, *datapath*) é explicitamente implementado. É

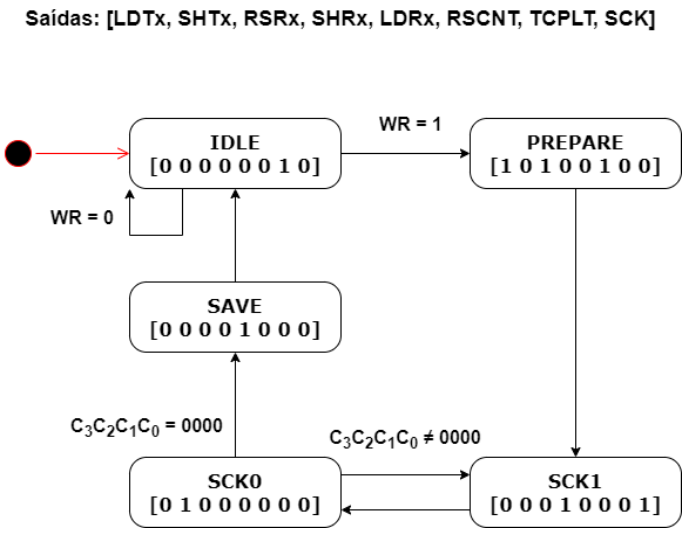
importante notar que os sinais MISO e MOSI, que fazem parte da interface física SPI, estão incluídos diretamente nesse fluxo do *datapath*.



3. Há uma sexta entidade, a `StateMachine`, que atua como a unidade de controle principal. Ela é uma **máquina de estados finitos** (em inglês, *Finite State Machine*) pura, gerenciando as transições entre seus dois sinais de estado, `state` e `next_state`, através de uma lógica de transição interna e as entradas `WR` e `Z`. Essa `StateMachine` gera um vetor de controle de 8 *bits* (`output`), que é mapeado para o sinal `Ctrl_internal` da entidade `MasterSPI`. Os *bits* desse vetor controlam diretamente todos os elementos do *datapath* previamente especificados no `MasterSPI`. Eles são: `LDTx`, `SHTx`, `RSRx`, `SHRx`, `LDRx` e `RSCNT`. Além desses, a FSM também é responsável por gerar os sinais externos `TCPLT` (do inglês *Transfer Complete*) e `SCK` (do inglês *Serial Clock*). Esses sinais estão organizados em um vetor com 8 sinais na seguinte ordem (do *bit* mais significativo para o menos significativo) na descrição:

[LDTx, SHTx, RSRx, SHRx, LDRx, RSCNT, TCPLT, SCK]

A figura abaixo mostra o diagrama de estados da FSM implementada na entidade `StateMachine` em VHDL.



Todos os elementos do *datapath* são síncronos, ou seja, os sinais de controle (LDRx, LDTx, SHTx, SHRx, RSRx, RSCNT) são processados na borda de **descida** do *clock*. Os sinais de controle são ativos em nível alto. A FSM, por sua vez, usa *flip-flops* sensíveis à borda de **subida** do *clock*. Assim, a cada ciclo de *clock*, primeiro a FSM atualiza o estado (borda de subida) e depois o *datapath* processa os sinais de controle (borda de descida).

4. Pelo lado da transmissão, temos no caminho de dados um registrador comum de 8 *bits* (**TxReg**), que recebe o dado a ser transmitido. Ele é carregado na borda de **subida** de *clock*, sendo que o *clock* deste registrador é ligado ao sinal ***WR*** (a única exceção no *datapath*, todos os outros elementos são ligados ao *clock* do controlador, respondendo a borda de descida), sendo sua entrada **LOAD** mantida em 1. Quando ***WR*** vai a 1, o dado a ser transmitido é imediatamente carregado no registrador de entrada, e na próxima borda de subida do *clock* o estado é atualizado. Se este registrador usasse o mesmo *clock* dos demais, mesmo que respondendo na borda de subida, seria necessário um estado adicional para concluir a carga do dado no registrador. Esta é uma estratégia usada para reduzir o número de estados necessários, porém deve ser usada com cautela.

Temos ainda um *shift register* do tipo paralelo-serial (**TxShiftReg**), que recebe o dado de transmissão paralelo e o serializa, com a carga do dado paralelo sendo comandada pelo sinal interno **LDTx** e o deslocamento de um *bit* à esquerda sob comando do sinal interno **SHTx**. Sua saída Q_7 (*bit* mais significativo) fornece o sinal **MOSI** da interface SPI.

5. Pelo lado da recepção, temos no caminho de dados inicialmente um *shift register* do tipo serial-paralelo (**RxShiftReg**), cuja entrada serial é o sinal **MISO** da interface SPI. Seu *reset* é controlado pelo sinal interno **RSRx** e a carga do *bit* de entrada para a saída Q_0 e deslocamento dos demais *bits* à esquerda é controlada pelo sinal interno **SHRx**.

Temos ainda um registrador simples que armazena o resultado da transferência e conversão serial-paralelo (**RxReg**), cuja carga de dados é comandada pelo sinal interno **LDRx**. Note que quando falamos em comandos, estes só são executados na borda de descida do *clock* do sistema.

Finalmente, há um contador de 4 *bits* (**C_0-C_3**) com seu *reset* controlado pelo sinal interno **RSCNT**. Este contador irá contar o número de *bits* transferidos.

6. Em relação à máquina de estados, segue-se uma descrição detalhada de cada estado:

- Estado **IDLE**: apenas o sinal **TCPLT** está ativo, e o controlador aguarda um sinal ***WR*** para iniciar a transferência bidirecional. Quando o sinal chega, muda para o estado **PREPARE**.
- Estado **PREPARE**: Ocorre uma preparação para o início da transferência. Na transição de **IDLE** para **PREPARE**, **TxReg** foi carregado com o dado a ser transmitido. Agora os sinais **LDTx**, **RSRx** e **RSCNT** são ativados, para carregar **TxShiftReg** com o dado a ser transmitido, zerar **RxShiftReg** e zerar o contador de *bits*. No próximo evento de *clock*, o estado muda para **SCK1**.
- Estado **SCK1**: Já temos o *bit* mais significativo da transmissão em **MOSI**, e, considerando que o sinal **\CS** estava ativo antes do início da transmissão, o *bit* mais significativo da recepção já foi colocado em **MISO**. É o momento de amostrar o primeiro *bit*, portanto o sinal **SCK** deve ter uma borda de subida. O sinal **SHRx** é ativado para carregar o primeiro *bit* da recepção em **RxShiftReg** e o sinal **SCK** vai para o valor 1. Note que na borda de descida, como o contador foi zerado mas o sinal de *reset* não está mais ativo, o contador passa a ter o valor 1. No próximo evento de *clock*, o estado muda para **SCK0**.
- Estado **SCK0**: É hora de mudar os *bits* em MOSI e MISO. O sinal **SHTx** é ativado, fazendo **TxShiftReg** deslocar um bit à esquerda, colocando o próximo *bit* em **MOSI**. O sinal **SCK** vai ao valor 0, sinalizando ao dispositivo *slave* que o *bit* em **MISO** deve ser trocado. Aqui existe uma decisão. Considere inicialmente que o valor do contador é 1. Se o valor do contador for diferente de zero, o próximo evento de *clock* muda o estado para **SCK1**. Aqui o contador é novamente incrementado, passando a ter o valor 2. Note que o valor do contador corresponde ao número de transições de **SCK** (tanto bordas de subida quanto de descida), e portanto para 8 *bits* devemos contar 16 transições.
- Os estados **SCK1** e **SCK0** se alternam enquanto o contador de transições avança até 15, que corresponde à última borda de subida (estado **SCK1**), e logo depois sofre um *overflow*, retornando ao valor zero na próxima transição (entrando no estado **SCK0**). Neste momento, como o valor do contador é igual a zero, a FSM. em vez de mudar para **SCK1** novamente, passa para o estado **SAVE**. Note que na primeira vez que a FSM entra no estado **SCK0**, o valor do contador já é 1, mudando para 2 antes da próxima borda de subida do *clock*, quando ocorre o teste de condição. Assim, apenas quando as 16 transições de **SCK** terminarem, o contador terá novamente o valor zero.
- Estado **SAVE**: Carrega o conjunto de *bits* recebidos em **MISO** no registrador **RxReg**, ativando o sinal **LDRx**. Depois, retorna a **IDLE**, colocando **TCPLT** em 1 novamente e completando a transferência

7. Acesse o *link* do EDA Playground, onde foi criado o projeto completo do controlador, denominado `MasterSPI`. Use o botão *Copy* para criar uma cópia em sua própria conta, e

navegue até seu próprio *Playground*, abrindo a cópia local. A seguir, vamos analisar a estrutura do arquivo de *design* e do arquivo de *testbench*.

8. Inicialmente são definidas as bibliotecas. A definição vale para a primeira entidade encontrada. Quando há várias entidades em um mesmo arquivo, as bibliotecas usadas devem ser definidas para cada entidade, e assim pode-se perceber a redefinição das bibliotecas para cada entidade definida no arquivo.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
```

9. Logo depois aparece a definição de **TxReg**.

```
4  -- ELEMENTOS DO DATAPATH
5  -- Entidade TxReg
6  entity TxReg is
7      Port (
8          clk   : in STD_LOGIC;
9          Tx_load  : in STD_LOGIC;
10         Tx_in  : in STD_LOGIC_VECTOR(7 downto 0);
11         Tx_out : out STD_LOGIC_VECTOR(7 downto 0);
12     );
13 end entity TxReg;
14
15 architecture Behavioral of TxReg is
16     begin
17         process(clk)
18         begin
19             if rising_edge(clk) then
20                 if Tx_load = '1' then
21                     Tx_out <= Tx_in;
22                 end if;
23             end if;
24         end process;
25 end architecture Behavioral;
```

A descrição está dividida em duas seções principais. A primeira, `entity TxReg`, define a interface externa do módulo, equivalente às portas de entrada e saída no esquemático apresentado no item 2. Isso inclui as entradas de dados `Tx_in` (correlacionadas a T0-T7 no esquemático), o sinal de clock `clk` (WR), o sinal de controle `Tx_load` (LOAD), e os sinais de saída `Tx_out`. Já a seção `architecture Behavioral` descreve o comportamento interno do `TxReg` como um registrador paralelo de 8 *bits* com carga síncrona habilitada.

Pela descrição, o comportamento é paralelo porque todos os 8 *bits*, representados pelo tipo de dados `STD_LOGIC_VECTOR(7 downto 0)`, são carregados e lidos simultaneamente através da atribuição `Tx_out <= Tx_in`. Ele é síncrono porque sua operação de carga é controlada pelo sinal de clock `clk`. Por fim, é habilitado porque o sinal de controle `Tx_load` (correspondente a `LOAD` no esquemático) determina quando essa carga pode ocorrer. Internamente, o sintetizador VHDL interpreta essa descrição para inferir oito *flip-flops* D (ou JK, dependendo das otimizações), um para cada *bit*, todos controlados pelo mesmo *clock* e sinal de habilitação de carga `Tx_load`.

De forma análoga, segue-se a descrição de `RxReg` com a mesma estrutura e lógica.

10. Em seguida, temos a descrição do registrador TxShiftReg que tem como interface externa 8 *bits* de entrada paralela `D` e 1 *bit* de saída `Q7` (MOSI no esquemático do item 2), além do sinal de `clk` (CLK), sinal de carga `LOAD` (LDTx) e um sinal de habilitação do deslocamento dos *bits* `SHIFT` (SHTx).

```
-- Entidade TxShiftReg
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity TxShiftReg IS
    Port (
        clk   : IN  STD_LOGIC;                -- Clock de entrada
        LOAD  : IN  STD_LOGIC;                -- Sinal de carga
        SHIFT : IN  STD_LOGIC;                -- Sinal de deslocamento
        D     : IN  STD_LOGIC_VECTOR(7 downto 0); -- Dados de entrada paralelos
        Q7    : OUT STD_LOGIC                 -- Bit de saída (Dado mais significativo)
    );
end entity TxShiftReg;

architecture Behavioral of TxShiftReg is
    -- Registrador interno de 8 bits iniciando com zero
    signal reg : STD_LOGIC_VECTOR(7 downto 0):= (others => '0');
    begin
        process(clk)
        begin
            if falling_edge(clk) then
                -- Carga paralela quando LOAD está em '1'
                if LOAD = '1' then
                    reg <= D;
                -- Deslocamento à esquerda quando SHIFT está em '1'
                elsif SHIFT = '1' then
                    reg <= reg(6 downto 0) & '0';  -- Desloca à esquerda e insere '0' no bit menos
significativo
                end if;
            end if;
        end process;

        -- Saída do bit mais significativo
        Q7 <= reg(7);
end architecture Behavioral;

```

A seção `architecture Behavioral` detalha o comportamento de um registrador de deslocamento de 8 *bits*, operando de forma síncrona sob o controle de um sinal de *clock* `clk`. A funcionalidade central do circuito reside na lógica de controle de carga e deslocamento, implementada no processo sequencial `process`. O sinal `reg`, declarado como `signal reg : STD_LOGIC_VECTOR(7 downto 0) := (others => '0')`, atua como registrador de 8 *bits*, armazenando dados temporários. O sintetizador VHDL infere um registrador para `reg` ao detectar uma atribuição sequencial (`reg <= D`) controlada pelo teste de borda de *clock*, `if falling_edge(clk) then`. Essa condição assegura que as operações sobre `reg` ocorram sincronizadamente na borda de descida do sinal de *clock*, garantindo o funcionamento correto do circuito síncrono.

Quando o sinal `LOAD` está ativo, o conteúdo do barramento de entrada `D` é carregado no `reg` de forma paralela. Se `LOAD` não estiver ativo e o sinal `SHIFT` estiver, o valor em `reg` é deslocado um *bit* para a esquerda (utilizando `reg(6 downto 0)`), com a inserção de um '0' no *bit* menos significativo (através da concatenação pela direita, `&'0'`) para que o

sintetizador gere uma entrada válida de 8 *bits* para `reg` em cada pulso de *clock*. A utilização da declaração sequencial `if ... then ... elsif ...` estabelece uma prioridade implícita para o sintetizador VHDL: a carga paralela sempre acontece antes do deslocamento. Por fim, a saída `Q7` reflete o valor do *bit* mais significativo de `reg` após o próximo ciclo de *clock*.

11. Após a descrição do TxShiftReg, temos o RxShiftReg. Este componente possui como entradas os sinais `clk` (correspondente a CLK no esquemático do item 2), `RESET` (RSRx), `SHIFT` (`SHRx`), e `D` (MISO). Sua única saída é o sinal `Q` (R0-R7), um barramento de 8 *bits*.

```
87  -- Entidade RxShiftReg
88  library IEEE;
89  use IEEE.STD_LOGIC_1164.ALL;
90
91  entity RxShiftReg IS
92      Port (
93          clk   : IN  STD_LOGIC;                -- Clock de entrada
94          RESET : IN  STD_LOGIC;                -- Sinal de reset
95          SHIFT : IN  STD_LOGIC;                -- Sinal de deslocamento
96          D     : IN  STD_LOGIC;                -- Entrada serial
97          Q     : OUT STD_LOGIC_VECTOR(7 downto 0) -- Saída paralela
98      );
99  end entity RxShiftReg;
100
101 architecture Behavioral of RxShiftReg is
102     signal reg : STD_LOGIC_VECTOR(7 downto 0); -- Registrador interno de 8 bits
103     begin
104         process(clk)
105         begin
106             if falling_edge(clk) then
107                 -- Se RESET estiver ativo, zera o registrador
108                 if RESET = '1' then
109                     reg <= (others => '0');
110                 -- Realiza o deslocamento à esquerda se SHIFT estiver ativo
111                 elsif SHIFT = '1' then
112                     reg <= reg(6 downto 0) & D;  -- Desloca à esquerda e insere o bit de entrada 'D'
no bit menos significativo
113                 end if;
114             end if;
115         end process;
116     -- Atribuição da saída paralela ao registrador interno
117     Q <= reg;
118 end architecture Behavioral;
119
```

A seção `architecture Behavioral` descreve um registrador de deslocamento de 8 *bits*, operando de forma síncrona com o sinal de relógio. Sua funcionalidade principal é controlada pelos sinais de `RESET` e `SHIFT`. Como na descrição de `TxShifReg`, o sinal `reg` atua como um registrador de 8 *bits*, armazenando dados temporários. Dentro do processo sensível ao `clk` (`process(clk)`), as operações ocorrem na borda de descida do *clock*. Se o sinal `RESET` estiver ativo (‘1’), o registrador `reg` é zerado. Caso contrário, se `SHIFT` estiver ativo (‘1’), o conteúdo de `reg` é deslocado um *bit* para a esquerda (`reg(6 downto 0)`) com o *bit* da entrada `D` sendo inserido na posição menos significativa (através da concatenação pela direita `&D`). A saída `Q` reflete continuamente o valor completo de 8 *bits* armazenado em `reg`. Após 8 pulsos de *clock* com `SHIFT` ativo e `RESET` inativo, um *byte* completo da entrada `D` será carregado serialmente no registrador, ficando disponível na saída `Q`.

12. Na sequência, temos a descrição de `Counter4Bit` (Contador no esquemático do item 2). A interface externa deste contador inclui o sinal de entrada `clk` (CLK), o sinal `RESET` (RSCNT) e dois sinais de saída `C` de 4 *bits* (C0-C3) e `Z` de 1 *bit*.

```
-- Entidade Contador de 4 bits
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Counter4Bit IS
    PORT (
        clk   : IN  STD_LOGIC;                 -- Clock de entrada
        RESET : IN  STD_LOGIC;                 -- Sinal de reset
        C     : OUT STD_LOGIC_VECTOR(3 downto 0); -- Saída do contador de 4 bits
        Z     : OUT STD_LOGIC                  -- Saída que indica quando o contador está em 0
    );
end entity Counter4Bit;

architecture Behavioral of Counter4Bit is
    signal count : STD_LOGIC_VECTOR(3 downto 0) := "0000"; -- Contador interno de 4 bits,
inicializado em 0
    begin
        process(clk)
        begin
            if falling_edge(clk) then
                -- Se RESET estiver ativo, zera o contador
                if RESET = '1' then
                    count <= "0000";
                else
                    -- Incrementa o contador
                    count <= count + 1;
                end if;
            end if;
        end process;
    -- Atribuição da saída do contador
    C <= count;
    -- Atribuição da saída Z, que indica se o contador está em zero
    Z <= '1' when count = "0000" else '0';
end architecture Behavioral;
```

A seção `architecture Behavioral` descreve um contador síncrono de 4 *bits*. Ele utiliza um sinal interno chamado `count`, inicializado em `0000`, para armazenar o valor atual da contagem. Sua operação é sincronizada pela borda de descida do sinal de *clock* (`clk`). Se o sinal `RESET` estiver ativo (‘1’), o contador é zerado. Caso contrário, o valor de `count` é incrementado em um a cada pulso de *clock*. A saída `C` reflete o valor atual de 4 *bits* do contador. Além disso, há uma saída `Z` que se torna ‘1’ apenas quando o contador está em `0000`, indicando que ele atingiu zero, e ‘0’ em qualquer outro estado.

13. O último componente do controlador, não presente explicitamente no esquemático do item 2, é a entidade `StateMachine`. Ela é responsável por implementar o diagrama de estados mostrado no mesmo item. Em sua interface externa, a `StateMachine` recebe o sinal de `clk` e o sinal `Z`, que indica o término de um ciclo de contagem de 16 pulsos pelo contador. Há também o sinal `WR`, utilizado para carregar um *byte* a ser transmitido. Por fim, a `StateMachine` possui uma saída de 8 *bits*, `output` (referenciada como `R0-R7` no esquemático do item 2), que corresponde ao vetor de sinais de controle `[LDTx, SHTx, RSRx, SHRx, LDRx, RSCNT, TCPLT, SCK]` detalhado no mesmo item.

```vhdl
-- ENTIDADE DA MAQUINA DE ESTADOS FINITOS
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity StateMachine is
    Port ( clk    : in STD_LOGIC;
           WR     : in STD_LOGIC;
           Z      : in STD_LOGIC;
           output : out STD_LOGIC_VECTOR(7 downto 0)
    );
end entity StateMachine;
```

```vhdl

architecture Behavioral of StateMachine is
    -- Definindo os estados
    type state_type is (IDLE, PREPARE, SCK1, SCK0, SAVE);
    signal state, next_state : state_type;

    -- Definindo o vetor de saída
    signal out_reg : STD_LOGIC_VECTOR(7 downto 0);

    begin
        -- Processo de máquina de estados
        process(clk)
        begin
            if rising_edge(clk) then
                state <= next_state;
            end if;
        end process;

    -- Atribuindo a saída para o vetor de saída
    output <= out_reg;

    -- Definindo a lógica de transição de estados e geração das saídas
    process(state, WR, Z)
    begin
        case state is
            when IDLE =>
                out_reg <= "00000010";  -- Saída inicial no estado IDLE
                if WR = '1' then
                    next_state <= PREPARE;
                else
                    next_state <= IDLE;
                end if;

            when PREPARE =>
                out_reg <= "10100100";  -- Valor de saída no estado PREPARE
                next_state <= SCK1;     -- Transição incondicional para SCK1

            when SCK1 =>
                out_reg <= "00010001";  -- Valor de saída no estado SCK1
                next_state <= SCK0;     -- Transição incondicional para SCK0

            when SCK0 =>
                out_reg <= "01000000";  -- Valor de saída no estado SCK0
                if Z = '1' then
                    next_state <= SAVE; -- Transição para SAVE se Z = 1
                else
                    next_state <= SCK1; -- Transição para SCK1 se Z = 0
                end if;

            when SAVE =>
                out_reg <= "00001000";  -- Valor de saída no estado SAVE
                next_state <= IDLE;     -- Transição incondicional para IDLE

            when others =>
                out_reg <= "00000010";  -- Valor de saída por padrão (IDLE)
                next_state <= IDLE;     -- Retorna para IDLE em caso de erro
        end case;
    end process;
end architecture Behavioral;
```

A seção `architecture Behavioral` descreve uma máquina de estados finita que controla uma sequência de operações baseada em seus estados internos e entradas externas. O sinal `out_reg` atua como um registrador de 8 *bits* para armazenar o vetor de sinais de controle em cada estado. Sincronizada pela borda de subida do *clock*, a máquina transita entre cinco estados (IDLE, PREPARE, SCK1, SCK0, SAVE). Em cada estado, ela gera um vetor de controle de 8 *bits* (`out_reg`). As transições são determinadas por sinais de entrada como `WR` (para iniciar uma operação a partir do estado `IDLE`) e `Z` (para sinalizar a conclusão de um ciclo no estado `SCK0`), ou são incondicionais, guiando o fluxo da sequência. A saída `output` espelha o vetor de controle gerado.

Note a descrição de dois processos concorrentes, `process(clk)` e `process(state,WR,Z)`, para representar a máquina de estados finitos. Essa é uma prática de *design* muito comum e altamente recomendada, conhecida como "*Two-Process FSM*":

- `process(clk)` é um **processo puramente síncrono** de registrador de estados, sensível apenas ao sinal de *clk* e responsável por registrar o próximo estado (`next_state`) no estado atual (`state`) na borda ativa do *clock* (neste caso, `rising_edge(clk)`).
- `process(state, WR, Z)` é um **processo** sensível às mudanças nos sinais de entrada que determinam o próximo estado (`state`, `WR`, `Z`). Como não é sensível ao *clock*, a lógica dentro dele é **combinacional**. Ele é responsável por calcular o `next_state` (lógica de transição de estados) com base no `current_state` (`state`) e nas entradas (`WR`, `Z`) e gerar os valores de saída (`out_reg`) com base no `estado corrente` (`state`).

Essa separação ajuda o sintetizador a inferir de forma mais eficiente os *flip-flops* para o estado e a lógica combinacional para as transições e saídas, resultando em um *hardware* mais otimizado em termos de área e *timing*, além de reduzir a probabilidade de criar lógica com *glitches* ou caminhos de *timing* problemáticos que podem surgir quando a lógica de registrador e a lógica combinacional estão emaranhadas.

14. Após as definições das 6 entidades que compõem o controlador (5 no *datapath* mais a FSM), temos a definição da entidade que engloba todos os itens, denominada `MasterSPI`. Sua porta, apresentada no item 1, apresenta todos os sinais de entrada e saída do controlador. Na arquitetura, inicialmente devem ser declarados os **componentes** a serem usados. Cada componente é semelhante aos **protótipos** das funções em C, descrevendo sua interface com os demais elementos do código.

```
243 architecture RTL of MasterSPI is
244     -- Declara componente TxReg
245     component TxReg
246         Port (
247             clk   : in STD_LOGIC;
248             Tx_load  : in STD_LOGIC;
249             Tx_in  : in STD_LOGIC_VECTOR(7 downto 0);
250             Tx_out : out STD_LOGIC_VECTOR(7 downto 0);
251         );
252     end component;
253 
254     -- Declara componente TxShiftReg
255     component TxShiftReg
256         Port (
257             clk   : IN  STD_LOGIC;
258             LOAD  : IN  STD_LOGIC;
259             SHIFT : IN  STD_LOGIC;
260             D     : IN  STD_LOGIC_VECTOR(7 downto 0);
261             Q7    : OUT STD_LOGIC
262         );
263     end component;
264 
265     -- Declara componente RxShiftReg
266     component RxShiftReg
267         Port (
268             clk   : IN  STD_LOGIC;
269             RESET : IN  STD_LOGIC;
270             SHIFT : IN  STD_LOGIC;
271             D    : IN  STD_LOGIC;
272             Q     : OUT STD_LOGIC_VECTOR(7 downto 0)
273         );
274     end component;
275 
276     -- Declara componente RxReg
277     component RxReg
278         Port (
279             clk   : in STD_LOGIC;
280             Rx_load  : in STD_LOGIC;
281             Rx_in  : in STD_LOGIC_VECTOR(7 downto 0);
282             Rx_out : out STD_LOGIC_VECTOR(7 downto 0);
283         );
284     end component;
285 
286     -- Declara componente contador de bits
287     component Counter4Bit
288         Port (
289             clk   : IN  STD_LOGIC;
290             RESET : IN  STD_LOGIC;
291             C     : OUT STD_LOGIC_VECTOR(3 downto 0);
292             Z     : OUT STD_LOGIC
293         );
```

15. Após a declaração dos componentes, são criados os sinais internos para a conexão entre os componentes. Temos vetores de 8 *bits* para as ligações entre TxReg e TxShiftReg (`Data_Tx`), entre RxShiftReg e RxReg (`Data_Rx`), e entre os sinais de controle gerados pela FSM e os demais elementos (`Ctrl_internal`). Além disso, há um sinal `zero` para conectar o indicador de zero do contador à FSM.

```
306     -- Sinais internos para interconexao
307     signal Ctrl_internal : STD_LOGIC_VECTOR (7 downto 0); -- Vetor dos 8 sinais internos
308     signal Data_Tx : STD_LOGIC_VECTOR (7 downto 0); -- Vetor dos dados de transmissao
309     signal Data_Rx : STD_LOGIC_VECTOR (7 downto 0); -- Vetor dos dados de recepcao
310     signal zero : STD_LOGIC; -- sinal interno de contador igual a zero
311 
```

16. Na sequência são definidas as **instâncias** dos componentes. Cada instância tem um nome (**Ux**) e define o componente correspondente, bem como o ***Port map***, que apresenta as conexões de cada entrada e saída do componente. Note que pode-se realizar conexões aos *bits* individuais do vetor *Ctrl_internal*. Note ainda que no contador, as saídas com o valor de contagem não são usadas, sendo mapeadas a uma conexão ***open***.

```
312 begin
313     -- Instância do TxReg
314     U1: TxReg
315         Port map (
316             clk => clk,
317             Tx_load => WR,
318             Tx_in => TxD,
319             Tx_out => Data_Tx
320         );
321 
322     -- Instância do TxShiftReg
323     U2: TxShiftReg
324         Port map(
325             clk => clk,
326             LOAD => Ctrl_internal(7), -- Sinal LDTx
327             SHIFT => Ctrl_internal(6), -- Sinal SHTx
328             D => Data_Tx,
329             Q7 => MOSI
330         );
331 
332     -- Instância do RxShiftReg
333     U3: RxShiftReg
334         Port map(
335             clk => clk,
336             RESET => Ctrl_internal(5), -- Sinal RSRx
337             SHIFT => Ctrl_internal(4), -- Sinal SHRx
338             D => MISO,
339             Q => Data_Rx
340         );
341 
342     -- Instância do RxReg
343     U4: RxReg
344         Port map(
345             clk => clk,
346             Rx_load => Ctrl_internal(3), -- Sinal LDRx
347             Rx_in => Data_Rx,
348             Rx_out => RxD
349         );
350 
351     -- Instância do contador de bits
352     U5: Counter4Bit
353         Port map(
354             clk => clk,
355             RESET => Ctrl_internal(2), -- Sinal RSCNT
356             C => open, -- Nao conectado
357             Z => zero
358         );
359 
360     -- Instância da FSM
361     U6: StateMachine
362         Port map(
363             clk => clk,
364             WR => WR,
365             Z => zero,
366             output => Ctrl_internal
367         );
```

17. Finalmente, são definidas as conexões entre o vetor interno de sinais de controle e as saídas externas **TCPLT** e **SCK**, pois estes são os sinais externos que não foram conectados em nenhuma das instâncias dos componentes.

```
369     -- Conexão direta dos sinais TCPLT e SCK da máquina de estados com as saídas
    correspondentes
370     TCPLT <= Ctrl_internal(1);
371     SCK <= Ctrl_internal(0);
```

18. No arquivo de *testbench* são definidos os sinais: Inicialmente, passam-se alguns ciclos de **clk** com **WR** em 0, mostrando que a FSM permanece em IDLE. Quando **WR** vai a 1, o valor 0xB4 (0b10110100) está presente em **TxD** (transmissão) e é carregado no registrador **TxReg**, sendo que logo depois o valor em **TxD** muda para 0x00. Isto foi feito para demonstrar que após a borda de subida de **WR**, o valor da entrada **TxD** pode mudar sem afetar a transferência de dados.

Na entrada MISO, são gerados os sinais para que o controlador receba o valor 0x69 (0b01101001). Note que os sinais gerados podem ser definidos a partir de eventos gerados pelo próprio sistema simulado (no caso a borda de descida de **SCK**), e não apenas a partir de tempos e outros sinais gerados no *testbench*.

```
48      -- Clock process: 10ns period (5ns high, 5ns low)
49      clk_process: process
50      begin
51          clk <= '0';
52          wait for 5 ns;
53          clk <= '1';
54          wait for 5 ns;
55      end process;
56
57      -- Test stimulus process
58      stimulus: process
59      begin
60          -- Wait for 2 clock cycles (20 ns) with all inputs at 0
61          wait for 20 ns;
62
63          -- Apply value 0xB4 (0b10110100) to TxD while clk is 0
64          TxD <= X"B4";
65          wait for 10 ns;
66
67          -- Wait for 2 more clock cycles (20 ns)
68          wait for 20 ns;
69
70          -- Set WR to 1 for one clock cycle, then back to 0
71          WR <= '1';
72          wait for 10 ns;
73          WR <= '0';
74
75          -- After WR returns to 0, reset TxD to 0x00
76          -- That proves that changing TxD after rising edge of WR does not change the byte
    sent by master
77          TxD <= X"00";
78
79          -- Change MISO on each falling edge of SCK
80          -- MISO value = 0x69 (0b01101001)
81          -- First '0' is already on MISO, wait next SCK falling edge
82          wait until SCK = '0';
83          MISO <= '1';
84
85          -- 0b01 already sent. Now send 0b101001
86          wait until SCK = '0';
87          MISO <= '1';
88          wait until SCK = '0';
89          MISO <= '0';
90          wait until SCK = '0';
91          MISO <= '1';
92          wait until SCK = '0';
93          MISO <= '0';
94          wait until SCK = '0';
95          MISO <= '0';
96          wait until SCK = '0';
97          MISO <= '1';
98
99          -- After 4 more clock cycles with SCK at 0, end the simulation
100         wait for 40 ns;
101         assert false report "End of simulation" severity note;
102         wait;
103     end process;
```

19. Ative "Open **EPWave** after run" na aba esquerda. Realize a simulação e veja o resultado. É possível usar o botão "*Get Signals*" para adicionar os sinais internos dos componentes, selecionando as instâncias dos mesmos. Selecione a instância **U6** e o sinal **output** ou **out_reg**. Será possível ver os vários sinais gerados pela FSM para controlar o *datapath*.

Organize os sinais para que **MISO**, **MOSI** e **SCK** fiquem juntos, pois assim pode-se apreciar o funcionamento da interface SPI. Lembre que no modo 0, os *bits* em **MOSI** e **MISO** são lidos na borda de subida de **SCK**, e mudam de valor enquanto **SCK** = 0.

20. Compare o sinal simulado com o sinal esperado do protocolo SPI. Veja os *bits* de **MOSI** em cada borda de subida de **SCK** e note o valor em **RxD** no final da transferência (quando **TCPLT** volta a 1).

21. Ao longo da descrição deste projeto, utilizamos três conceitos importantes de abstração de hardware: nível de transferência de registradores (RTL), nível de máquina de estados finitos (FSM) e nível de máquina de estados finitos com caminho de dados (FSMD). A implementação em VHDL do projeto atual foi feita em FSMD. Você saberia explicar a diferença entre esses três níveis de abstração? Se ainda não souber, não se preocupe; vamos explorar essas diferenças em breve.

FUNDAMENTOS TEÓRICOS

No desenvolvimento de sistemas embarcados, a complexidade dos projetos modernos exige uma abordagem colaborativa e integrada, conhecida como **co-projeto**. Essa metodologia reúne equipes multidisciplinares que trabalham juntas para otimizar a interação entre *hardware* e *software*, assegurando que cada componente funcione harmoniosamente dentro do sistema. O co-projeto é especialmente relevante na integração de microcontroladores, como o STM32, com módulos de *hardware* dedicados implementados em dispositivos de lógica programável.

Como vimos nos roteiros anteriores, os **microcontroladores** são pequenos "computadores" em um único *chip* que incluem um processador, memória e entradas e saídas. Eles são ótimos para tarefas específicas, como controlar um motor ou ler sensores, e são programados usando linguagens como C. Em contraste, **dispositivos de lógica programável** oferecem maior flexibilidade, permitindo a criação de circuitos digitais personalizados por meio da descrição da lógica do *hardware*, utilizando linguagens de descrição de *hardware* (em inglês, *Hardware Description Language* – HDL).

Os microcontroladores executam um conjunto fixo de instruções, realizando tarefas de forma eficiente e seguindo uma sequência de comandos que não pode ser alterada fisicamente, o que os torna ideais para aplicações que requerem controle simples e direto. Já os dispositivos de lógica programável consistem em matrizes de blocos lógicos configuráveis, possibilitando

a criação e ajuste de circuitos digitais de acordo com as necessidades do projeto. Essa flexibilidade permite que um único dispositivo de lógica programável execute múltiplas tarefas simultaneamente, alcançando um nível de paralelismo que os microcontroladores não conseguem oferecer, o que é vantajoso em projetos que demandam processamento intensivo.

Os co-projetos em sistemas embarcados podem se beneficiar do uso de ambas as tecnologias. Os microcontroladores permitem programação rápida e fácil, ideal para ajustes em tempo real durante o desenvolvimento, enquanto os dispositivos de lógica programável possibilitam a adaptação rápida das soluções a novos requisitos de *design*. Em um co-projeto, as equipes podem usar microcontroladores para gerenciar a lógica de controle e dispositivos de lógica programável para tarefas mais complexas ou processamento de dados em larga escala, resultando em sistemas mais robustos e eficientes. A distinção entre o *design* de microcontroladores e de dispositivos de lógica programável se baseia na natureza de cada um. O *design* de microcontroladores envolve a programação em linguagens de alto nível, onde se define a lógica sequencial que o dispositivo executará, focando na implementação de algoritmos e controle lógico. Já o *design* de dispositivos de lógica programável requer a descrição da lógica digital a ser implementada no *hardware*, utilizando HDLs, o que envolve a configuração física do dispositivo e a interconexão dos blocos lógicos.

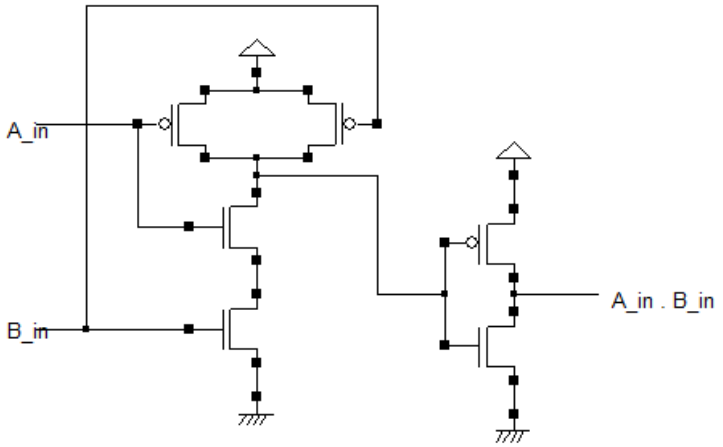
NÍVEIS DE ABSTRAÇÃO DE *HARDWARE*[1]

É fundamental especificar o nível de abstração do *hardware* ao descrever um circuito usando uma linguagem HDL. Isso ocorre porque, à medida que a densidade dos *chips* aumenta para centenas de milhões de transistores, torna-se impraticável, às vezes desnecessário, processar tantos dados diretamente. A abstração se torna uma estratégia essencial para gerenciar essa complexidade, simplificando o modelo do sistema ao focar apenas em características relevantes e ignorando detalhes irrelevantes, o que facilita a apresentação da informação essencial. Geralmente, distinguem-se, quanto ao tamanho dos blocos de construção fundamentais, quatro níveis de abstração no desenvolvimento de sistemas digitais: nível de transistores, nível de portas lógicas, nível de transferência de registradores (RT) e nível de processador. Consequentemente, ao mencionar uma descrição em HDL, como VHDL, é imprescindível especificar em qual nível ela está, pois cada nível implica uma quantidade diferente de detalhes de implementação e serve a propósitos distintos no fluxo de *design*.

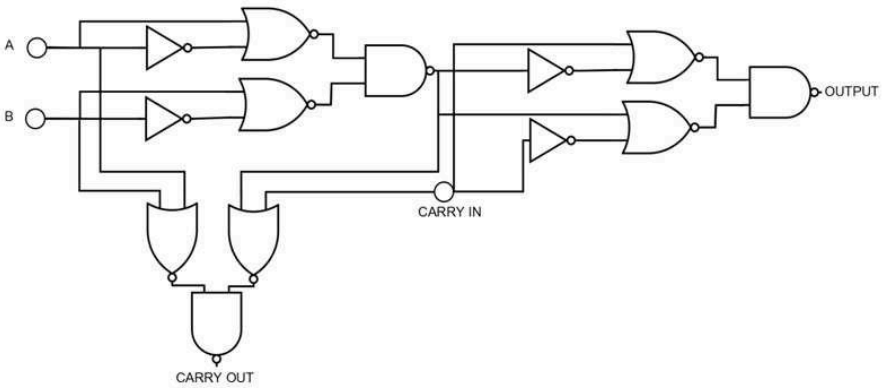
O nível de abstração mais próximo dos componentes eletrônicos fundamentais, como transistores e elementos passivos, é o **nível de transistores** (em inglês, *transistor level abstraction*). Nesse nível, os blocos de construção básicos são transistores, resistores, capacitores, entre outros. A descrição do comportamento geralmente é feita por um conjunto de equações diferenciais ou por algum tipo de diagrama de corrente-tensão. No nível de transistor, um circuito digital é tratado como um sistema analógico, no qual os sinais variam

[1] O conteúdo desta seção é baseado na Seção 1.4 do livro RTL Hardware Design Using VHDL com adaptações.

com o tempo e podem assumir qualquer valor dentro de um intervalo contínuo. A descrição física do nível de transistor compreende o *layout* detalhado dos componentes e suas interconexões, definindo essencialmente as máscaras das várias camadas e constituindo o resultado final do processo de *design*. A figura a seguir ilustra uma porta AND com 2 entradas representado a nível de transistores.

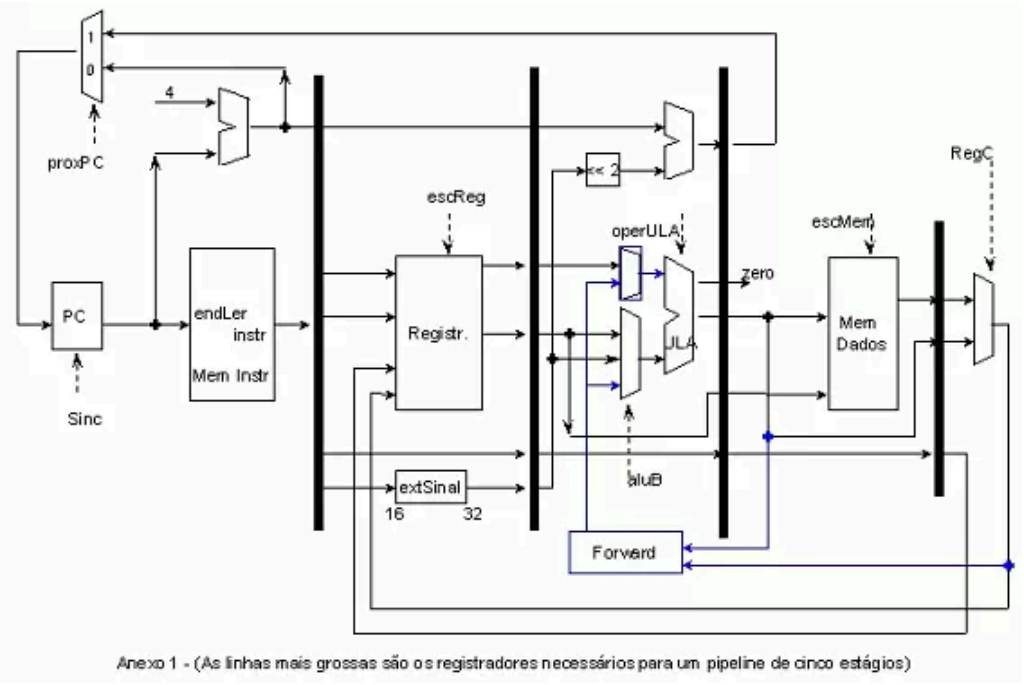


O próximo nível de abstração é o **nível de portas lógicas** (em inglês, *gate level abstraction*) . Os blocos de construção típicos incluem portas lógicas simples, como AND, OR, XOR, e multiplexadores 2-to-1 de 1 *bit*, além de elementos de memória básicos, como *latch* e *flip-flop*. Neste nível, consideramos apenas se a tensão de um sinal está acima ou abaixo de um limite, interpretando isso como lógica 1 ou 0, respectivamente. A descrição do comportamento de entrada e saída é feita por equações booleanas, convertendo um sistema contínuo em um sistema discreto e descartando equações diferenciais complexas. O timing é simplificado por meio de um único número, chamado de atraso de propagação, que indica o tempo necessário para que um sistema produza uma saída estável. A representação física neste nível refere-se à disposição das portas e ao roteamento das conexões. O tamanho do circuito pode ser descrito em termos da área ocupada ou pela contagem de portas, usando a área da porta NAND de duas entradas como unidade base, permitindo que a medição seja independente da tecnologia utilizada. A figura ilustra um somador completo representado a nível de portas lógicas.



No **nível de transferência de registradores** (em inglês, *register transfer level abstraction*), os blocos de construção básicos são módulos formados por portas simples, incluindo

unidades funcionais como somadores e comparadores, componentes de armazenamento como registradores, e elementos de roteamento de dados como multiplexadores. O termo "transferência de registradores" se refere a uma metodologia de projeto que especifica a operação do sistema com base na manipulação e transferência de dados entre registradores de armazenamento, como o projeto de um processador organizado como um *pipeline* de *n* estágios. A figura a seguir mostra um circuito de um processador a nível de RTL, organizado como um *pipeline* de 5 estágios com 4 "barreiras" de registradores de armazenamento para sincronizar os sinais.



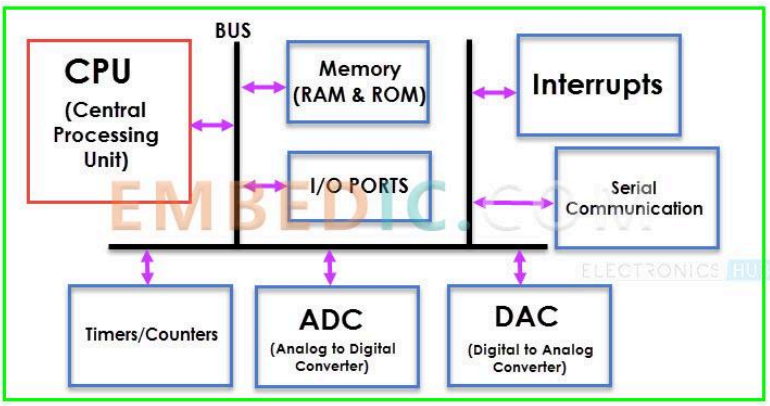
Anexo 1 - (As linhas mais grossas são os registradores necessários para um pipeline de cinco estágios)

No nível RTL, o funcionamento de um sistema digital é descrito em termos de transferências de dados entre registradores, que atuam como locais temporários de armazenamento, intercalados por operações combinacionais. A representação dos dados se torna mais abstrata nesse nível, com sinais frequentemente agrupados em vetores ou interpretados como tipos de dados especiais, como inteiros sem sinal. As operações sequenciais, que ocorrem ao longo do tempo, são sincronizadas por um sinal de *clock* comum, o qual funciona como um pulso de amostragem que determina quando os dados devem ser inseridos nos registradores, geralmente nas bordas de subida ou descida do *clock*.

Essa abordagem permite que o *timing* do sistema seja analisado em termos de ciclos de *clock*, ao invés de exigir o rastreamento detalhado de todos os atrasos de propagação no nível de porta. Para garantir confiabilidade, o período do *clock* deve ser suficientemente longo para que todos os sinais envolvidos se estabilizem antes do próximo disparo. A disposição física dos componentes nesse nível, conhecida como *floorplan*, ajuda a identificar o caminho crítico (mais lento) entre registradores, o que influencia diretamente a determinação do período de *clock* ideal. O circuito ou sistema descrito nesse nível de abstração é frequentemente denominado **processador** (processor) ou uma **propriedade intelectual** reutilizável (IP – *Intellectual Property*), e serve como base para a síntese automática de *hardware* digital.

Assim, o modelo RTL equilibra abstração funcional e viabilidade de implementação, tornando-se essencial no projeto eficiente de sistemas digitais complexos.

A abstração em **nível de processadores** (em inglês, *processor level abstraction*) é o nível que se refere à interação entre os processadores. Os blocos de construção básicos neste nível incluem processadores, módulos de memória, interfaces de barramento, entre outros, como mostra a figura a seguir. A descrição do comportamento do sistema pode ser representada de forma similar a um programa escrito em uma linguagem de baixo nível, como *assembly* ou C, incluindo a definição de etapas computacionais e interações entre os diversos módulos. Os sinais de controle e dados são agrupados e abstraídos em registradores de controle, configuração, dados e estado. Esses registradores gerenciam as operações internas e a comunicação com o sistema, algo semelhante ao que ocorre na **programação de um microcontrolador**. A medição de tempo é expressa em termos de uma etapa de computação, que é composta por um conjunto de operações definidas entre dois pontos de sincronização sucessivos. Um conjunto de instruções pode ser executado concorrentemente em *hardware* paralelo e trocar dados por meio de um protocolo de comunicação mediado por interrupções. Os eventos de interrupção são usados para sinalizar a necessidade de interação ou troca de informações, como a notificação de que um dado foi produzido por um processador e está pronto para ser consumido por outro. A disposição física de um sistema no nível de processadores é frequentemente referida como *floorplan*, que descreve o arranjo físico dos componentes do sistema, como núcleos de processamento, memórias e interconexões. No entanto, os componentes em um *floorplan* no nível de processadores geralmente são de maior escala e complexidade em comparação com os utilizados em nível RT.



A tabela a seguir resume as principais características de cada nível, listando os blocos de construção típicos, a representação de sinais, a representação de tempo, a descrição comportamental representativa e a descrição física representativa.

Table 1.2 Characteristics of each abstraction level

	Typical blocks	Signal representation	Time representation	Behavioral description	Physical description
Transistor	transistor, resistor	voltage	continuous function	differential equation	transistor layout
Gate	and, or, xor, flip-flop	logic 0 or 1	propagation delay	Boolean equation	cell layout
RT	adder, mux, register	integer, system state	clock tick	extended FSM	RT-level floor plan
Processor	processor, memory	abstract data type	event sequence	algorithm in C	IP-level floor plan

MÁQUINA DE ESTADOS FINITOS

Embora o desenvolvimento de sistemas digitais envolva diversos níveis de abstração, desde os transistores até a arquitetura de processadores, todos compartilham uma característica fundamental: seu comportamento pode ser descrito como uma sequência de estados e transições entre esses estados. No nível mais baixo, os circuitos digitais são implementados principalmente com **transistores** MOSFETs, que atuam como chaves controláveis, alternando entre dois estados de condução distintos. Essa comutação permite representar dois níveis lógicos bem definidos, geralmente associados aos valores binários ‘0’ (nível lógico baixo) e ‘1’ (nível lógico alto). No nível imediatamente superior, temos as **portas lógicas**, que combinam esses sinais binários para realizar operações booleanas. A partir delas, formam-se blocos mais complexos, como *flip-flops* e registradores, os quais introduzem o conceito de armazenamento de estado que é um elemento essencial nos circuitos sequenciais.

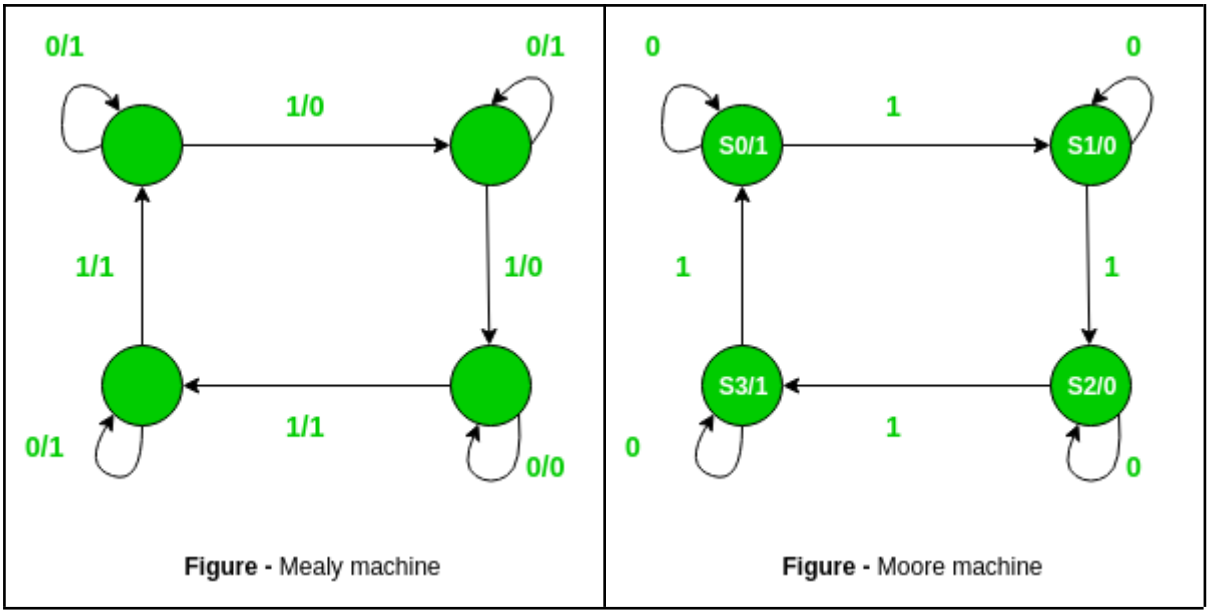
No nível **RTL**, o foco está na movimentação de dados entre registradores, coordenada por sinais de temporização e lógica de controle. O sistema é descrito por um conjunto de estados distintos, cada um representando uma configuração específica dos valores nos registradores e nos sinais de controle. As transições entre esses estados ocorrem em resposta a eventos como pulsos de *clock* ou alterações nas entradas do sistema. Por fim, no nível da **arquitetura do processador**, o comportamento do sistema é descrito como uma sequência de ciclos de execução de instruções, como busca (*fetch*), decodificação (*decode*) e execução (*execute*). Cada fase corresponde a um estado, e as transições entre essas fases são controladas por uma unidade lógica de controle. Assim, independentemente do nível de abstração considerado, o comportamento dos circuitos sequenciais e dos sistemas de controle lógico pode ser formalizado por meio de modelos matemáticos baseados em máquinas de estados.

Uma **máquina de estados** é um modelo matemático usado para representar sistemas que podem estar em um número finito de estados, respondendo a entradas externas e mudando de

estado de maneira específica, de acordo com essas entradas. Os estados representam as condições ou situações que o sistema pode estar em um dado momento, enquanto as transições determinam como o sistema se move de um estado para outro, dependendo das entradas recebidas. Esses sistemas podem ser simples, com poucos estados, ou complexos, com muitos estados e transições. Formalmente, uma máquina de estados M pode ser representada por um sextúplo $M = (\Sigma, \Gamma, Q, \delta, q_0, F)$, onde

- Σ representa um **alfabeto de entrada** que corresponde a um conjunto finito de símbolos que a máquina pode receber como entrada. Esses símbolos podem ser sinais digitais (como 0 e 1) ou valores analógicos.
- Γ é um **alfabeto de saída** que é um conjunto finito de símbolos que a máquina pode gerar como saída.
- Q é um conjunto de **estados** em que a máquina pode se encontrar durante sua operação.
- δ é um conjunto de **funções de transição** $\delta : Q \times \Sigma \rightarrow Q$ que definem o comportamento da máquina ao receber uma entrada, ou seja, define para qual estado a máquina transitará a partir de um estado atual, dado um símbolo de entrada. A função de transição pode ser representada de forma tabular, utilizando uma tabela de transições de estado, ou de forma gráfica, por meio de diagramas de transição.
- q_0 é um estado do conjunto Q, no qual a máquina começa a sua operação. É conhecido como **estado inicial**.
- F é um conjunto de **estados finais** que indicam que a máquina concluiu seu processamento ou atingiu um objetivo específico.

Quando o número de estados de uma máquina de estados é limitado, dizemos que ela é uma **máquina de estados finitos** (em inglês, *Finite State Machine* – **FSM**). Dependendo do tipo de saída, as FSMs podem ser classificadas em dois tipos principais: **Máquina de Mealy** e **Máquina de Moore**. Na **Máquina de Mealy**, a saída depende não apenas do estado atual, mas também das entradas recebidas, ou seja, a saída pode variar mesmo que o estado não mude, desde que haja uma mudança na entrada. Em contraste, na **Máquina de Moore**, a saída é determinada exclusivamente pelo estado atual, sendo independente das entradas. Essa distinção impacta diretamente o comportamento da máquina e a forma como ela responde a eventos, influenciando o *design* de sistemas digitais e sua implementação em *hardware*.



Fonte: Geeksforgeeks.

Modelar um sistema como uma máquina de estados finitos envolve uma combinação de conhecimento técnico e criatividade, e há alguns princípios que podem ajudar nesse processo, especialmente para quem está começando. O primeiro passo é identificar os diferentes **estados** que o sistema pode assumir. Esses estados precisam ser bem definidos, ou seja, cada estado deve ser único e, ao mesmo tempo, cobrir todas as possibilidades do sistema, sem incluir variações irrelevantes que não influenciem seu comportamento. Após definir os estados, é preciso determinar quais **entradas (**eventos**)** causam a **transição** de um estado para outro. Cada estado deve ter uma descrição clara de para onde pode se mover e em que momento isso ocorre. Em seguida, deve-se definir as **ações** associadas a cada estado e transição. Isso significa especificar o que o sistema faz quando entra em um estado, enquanto está nesse estado e também quando muda de estado. Além disso, é importante identificar o **estado inicial**, de onde o sistema começa, e o(s) **estado(s) final(is)**, caso haja. Outro aspecto a considerar é a possibilidade de falhas. Em qualquer sistema, é necessário prever **estados de erro**, que podem ocorrer por problemas como falhas na validação de entradas ou erros inesperados durante a execução. Por exemplo, em um sistema de pagamento, um estado de erro pode ser acionado caso haja um problema na comunicação com o banco. Se o sistema permitir que ele se recupere de falhas e retorne a estados anteriores, como em um processo de correção de erro, isso também deve ser claramente especificado.

A máquina de estados oferece uma descrição clara, precisa e unificada do comportamento de um sistema, garantindo que ele responda de forma controlada e previsível, apenas às condições adequadas. Isso evita transições indesejadas e comportamentos inesperados, aumentando a robustez, confiabilidade e eficiência da operação. Além disso, por ser uma abstração independente da implementação física, ela permite analisar o sistema em detalhes e prever todos os seus comportamentos, seja qual for o nível de abstração do *hardware* utilizado. No entanto, a máquina de estados é, por definição, um modelo abstrato. Sem uma forma visual de representação, entender como o sistema transita entre estados e responde a

eventos pode se tornar uma tarefa difícil e propensa a erros, especialmente em sistemas mais complexos.

Na comunidade de desenvolvimento de *software*, é amplamente reconhecido que diagramas gráficos comunicam ideias de forma mais rápida e eficaz do que descrições textuais extensas. A visualização torna modelos abstratos em algo tangível e mais fácil de compreender. Isso permite que desenvolvedores, *designers*, analistas e outras partes interessadas compreendam um sistema tanto em um nível macro quanto em detalhes específicos, dependendo do tipo de diagrama utilizado. Para acompanhar o comportamento sequencial de um sistema e verificar seu funcionamento de forma intuitiva e acessível, apresentamos dois recursos populares: tabelas de transição e diagramas de estados.

Tabelas de transição de estados

Para facilitar a visualização e compreensão dos estados e suas transições, as máquinas de estados são tradicionalmente representadas por **tabelas de transição de estados**. Essas tabelas são organizadas em colunas, sendo que a primeira geralmente lista todos os estados possíveis da máquina. Em seguida, há uma coluna para as entradas que podem ser recebidas e outra coluna que descreve os estados de destino correspondentes, ou seja, para cada combinação de estado e entrada, é indicado o próximo estado para o qual a máquina transitará. A tabela a seguir, extraída da Wikipedia, ilustra os estados de uma catraca típica, como aquelas encontradas em transportes públicos ou em eventos.

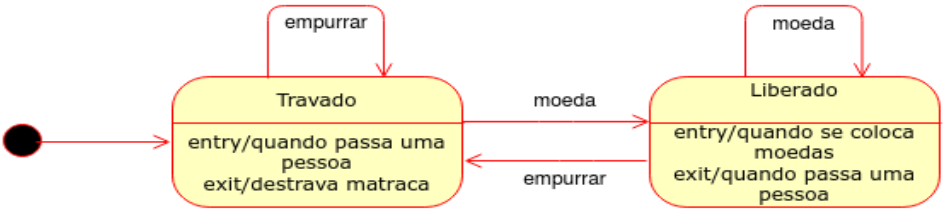
Current State	Input	Next State	Output
Locked	coin	Unlocked	Unlocks the turnstile so that the customer can push through.
	push	Locked	None
Unlocked	coin	Unlocked	None
	push	Locked	When the customer has pushed through, locks the turnstile.

Por exemplo, quando a catraca está no estado "travado", empurrar o braço da catraca não altera o seu estado. Da mesma forma, quando a catraca está no estado "liberado", inserir moedas adicionais não provoca nenhuma mudança. Esse comportamento, em que a saída depende apenas do estado atual e não das entradas, caracteriza uma máquina de Moore. Por outro lado, eventos como a inserção de moedas ou a ativação do braço da catraca são entradas que devem provocar transições entre os estados, alterando as saídas. Esse comportamento é característico de uma máquina de Mealy, onde a saída depende tanto do estado atual quanto da entrada recebida.

UML: Uma linguagem gráfica

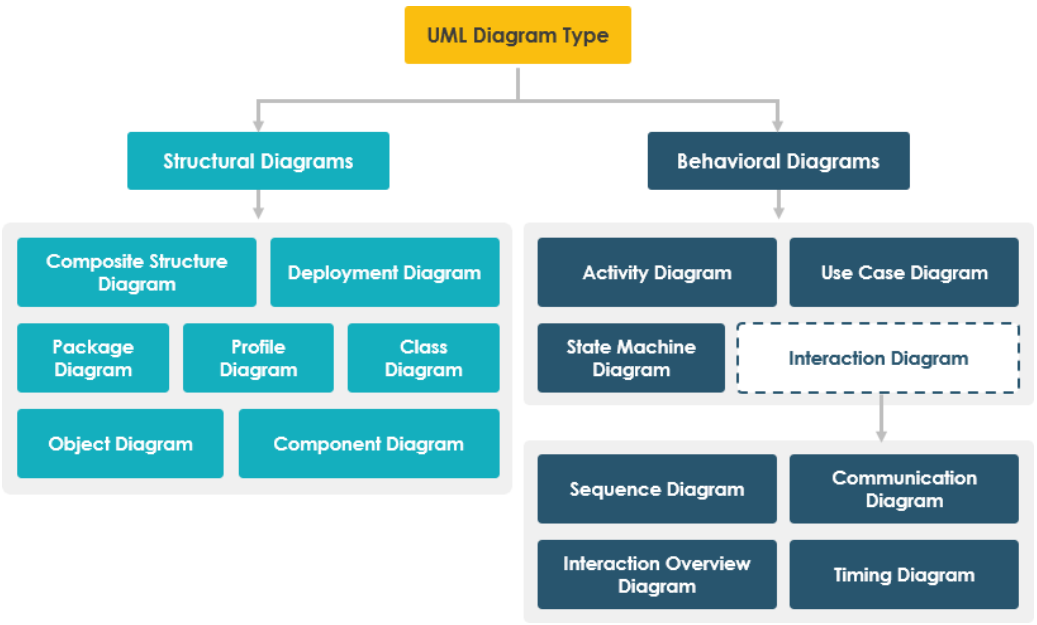
A **UML** (do inglês *Unified Modeling Language*) é uma linguagem padrão que oferece uma variedade de diagramas para modelar diferentes aspectos de um sistema. Entre eles, o

diagrama de máquina de estados é particularmente útil, pois descreve os estados de um objeto e as transições entre esses estados em resposta a eventos ou condições. Ao utilizar diagramas de estados da UML para representar máquinas de estados finitos (em inglês, *Finite State Machines* – FSMs), facilita-se significativamente o processo de projetar, compreender e validar o funcionamento de sistemas digitais descritos em linguagens como VHDL. A seguir, apresentamos um diagrama UML de máquina de estados que corresponde à tabela de transição de estados de uma catraca. Observe que as ações associadas a cada estado estão detalhadas nas caixas que representam os estados "Travado" e "Liberado".



No contexto de desenvolvimento de *software* ou *hardware*, como no uso de VHDL para a descrição de circuitos digitais, a UML e seus diagramas de estado auxiliam no alinhamento entre a implementação da máquina de estados e a visão abstrata do sistema, tornando o processo de desenvolvimento mais robusto e menos propenso a erros.

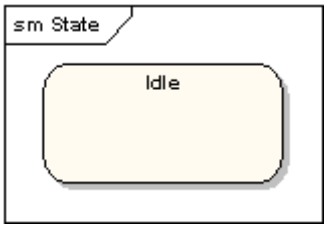
A UML (do inglês *Unified Modeling Language*), um padrão desenvolvido pela OMG (do inglês *Object Management Group*), é uma linguagem de modelagem visual amplamente utilizada no desenvolvimento de *software* e sistemas, permitindo representar a estrutura, o comportamento e as interações de um projeto de forma clara e padronizada. Na UML 2.2, existem 14 tipos de diagramas UML que abordam diferentes aspectos de um sistema, incluindo a modelagem estrutural do objeto de interesse, por meio de diagramas como os de classes e componentes, e a modelagem comportamental, através de diagramas como os de máquina de estados e atividades.



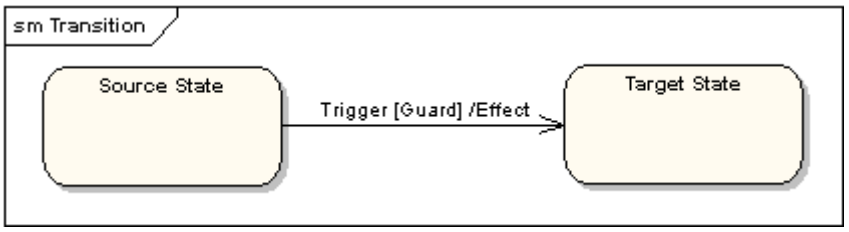
O **diagrama de estados UML**, especificamente, é utilizado para descrever o comportamento dinâmico de um sistema, ilustrando como um objeto ou componente muda de estado em resposta a eventos. Este diagrama é particularmente útil para modelar sistemas de controle, processos que dependem de sequências de eventos ou máquinas de estados finitos. Ele é composto por estados, pseudo-estados, transições, eventos e ações, permitindo uma visualização clara e intuitiva das regras de transição de um sistema. As máquinas de estados UML combinam características das máquinas de Mealy e das máquinas de Moore. Assim como nas máquinas de Mealy, elas podem disparar ações baseadas tanto no estado atual quanto no evento que aciona a transição. Por outro lado, como nas máquinas de Moore, também é possível associar ações de entrada e saída aos próprios estados, em vez de às transições entre eles.

Para representar e controlar fluxos complexos em sistemas dinâmicos, onde as transições entre estados podem ser condicionais, paralelas ou exigir pontos de início e término bem definidos, a UML inclui os **pseudo-estados**. Esses elementos especiais não representam estados reais do sistema, mas sim mecanismos que facilitam a gestão do fluxo de transições, tornando o modelo mais organizado. A seguir, apresentaremos as notações gráficas mais comuns na representação de máquinas de estados na UML.

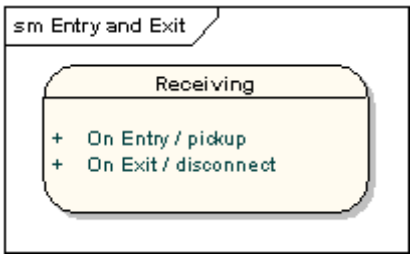
Um estado é representado por um retângulo com cantos arredondados, contendo o nome do estado em seu interior. Dentro desse retângulo, é possível incluir ações associadas ao estado, como ações de entrada (denotadas por `On Entry/`), de saída (denotadas por `On Exit/`) e ações do estado (denotadas por `do/`).



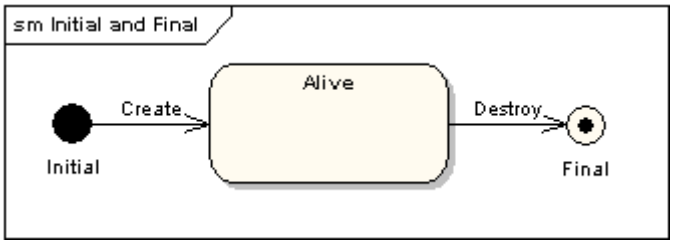
As transições entre estados são representadas por setas, acompanhadas de rótulos que especificam as entradas que causam as transições (`Trigger`), as condições que devem ser atendidas para que as transições ocorram (`Guard`) e o efeito que a transição tem no estado (`Effect`).



Caso o estado de destino tenha várias transições chegando até ele e todas compartilhem o mesmo efeito, é mais eficiente associar esse efeito diretamente ao estado, em vez de repetí-lo em cada transição. Isso pode ser feito definindo uma ação de entrada usando o rótulo "`On Entry/`" no estado de destino. Da mesma forma, se a saída de um estado implica em mudanças nas transições ou efeitos do próximo estado, pode-se definir uma ação de saída com o rótulo "`On Exit/`". O diagrama abaixo ilustra um estado com uma ação de entrada e uma ação de saída associadas.

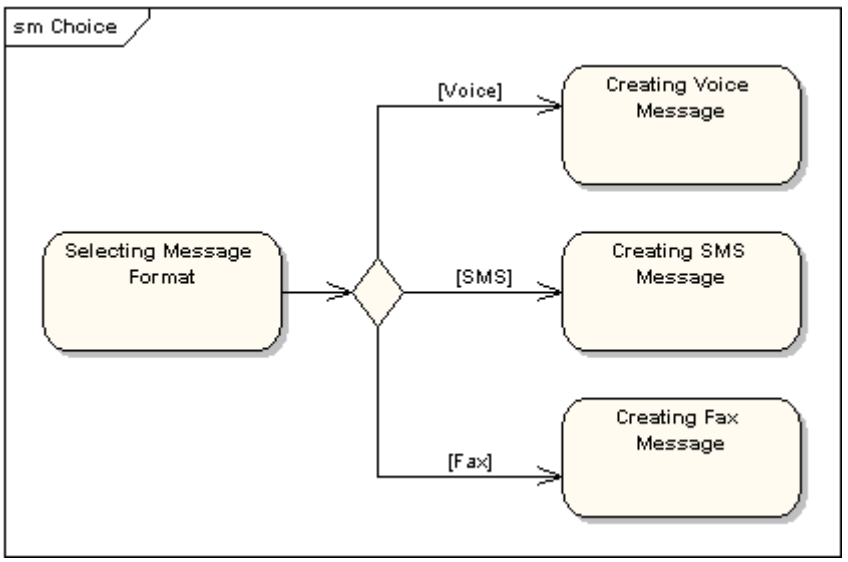


O estado inicial é representado por um círculo preto preenchido e pode ser rotulado com um nome. O estado final é representado por um círculo com um ponto dentro e também pode ser rotulado com um nome.

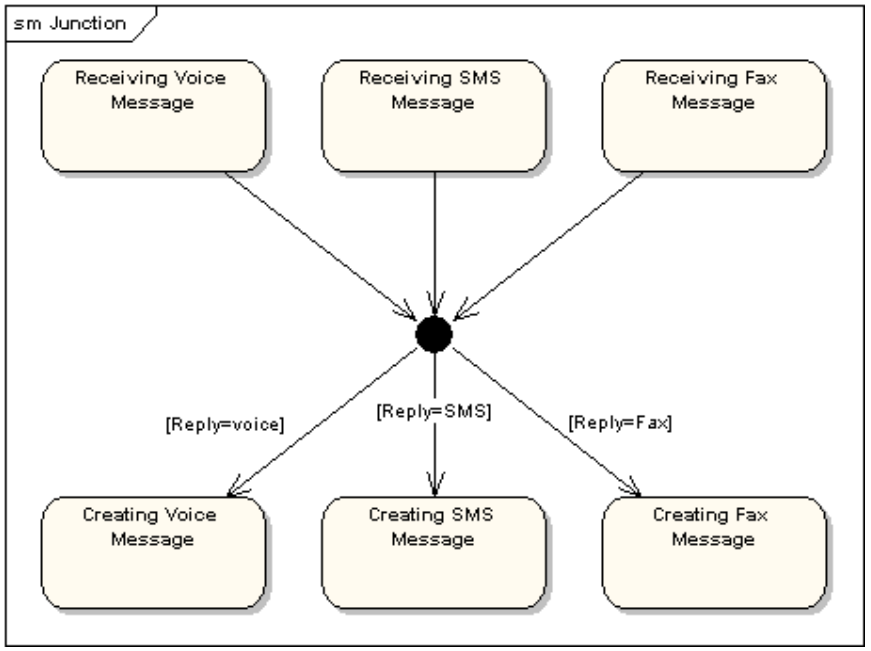


Entre os pseudo-estados, destacam-se aqueles que representam fluxos de decisão, junção e transições alternativas. O pseudo-estado de escolha é representado por um losango, com uma

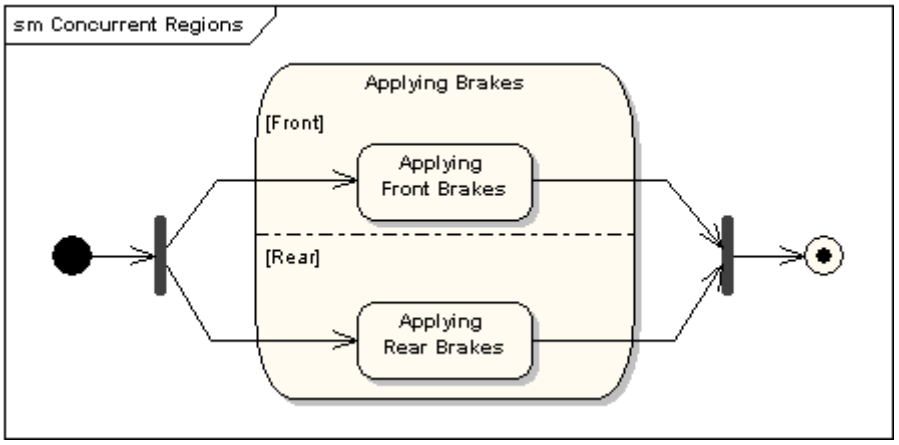
transição de entrada e duas ou mais transições de saída. O diagrama a seguir ilustra que, após o pseudo-estado de escolha, o estado alcançado depende do formato da mensagem selecionada durante a execução do estado anterior.



Os pseudo-estado de junção são usados para combinar múltiplas transições. Uma junção pode ter uma ou mais transições de entrada e uma ou mais transições de saída. As junções não possuem um significado específico por si mesmas. Quando uma junção divide uma transição de entrada em várias transições de saída, ela realiza uma **ramificação condicional estática** (paralela), ao contrário do pseudo-estado de escolha, que realiza uma **ramificação condicional dinâmica** (sequencial).



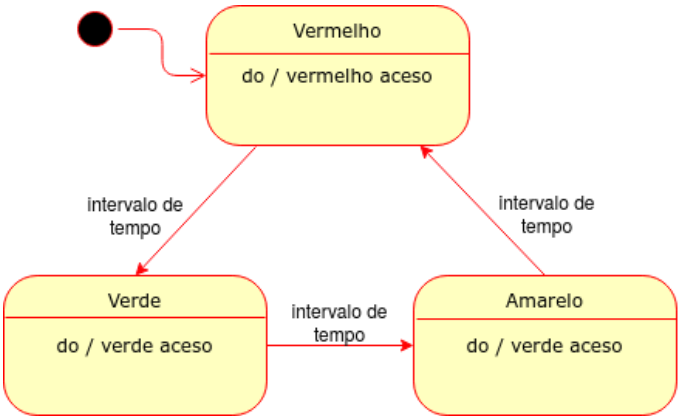
Por fim, temos ainda o pseudo-estado de transição alternativa (em inglês, *fork*) que é oposto de um pseudo-estado de junção. Ele é representado por uma **linha vertical** que divide uma única transição em várias ramificações. O *fork* é usado para criar transições paralelas, onde diferentes caminhos podem ser seguidos ao mesmo tempo a partir de um mesmo estado.



Segeu-se um exemplo que descreve o comportamento de um semáforo, que alterna entre estados em intervalos de tempo fixos. Este comportamento pode ser modelado como uma máquina de Moore. Isso ocorre porque a saída (a cor do semáforo) depende exclusivamente do estado atual, e as transições entre os estados acontecem de acordo com os períodos de tempo predefinidos. A tabela a seguir apresenta as transições de estados, detalhando as ações de cada estado, que consistem em manter uma cor específica acesa, e as mudanças de cor que ocorrem a cada transição.

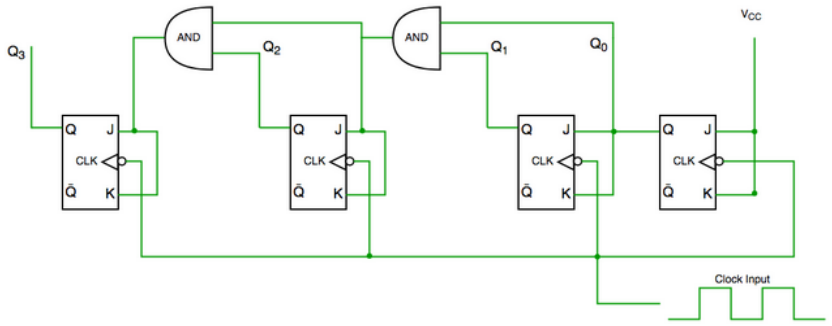
Estado Atual	**Entrada**	**Estado Seguinte**	**Saída**
Vermelho	-	Verde	Vermelho
Verde	-	Amarelo	Verde
Amarelo	-	Vermelho	Amarelo

A seguir, é apresentado o diagrama de máquina de estados na UML, que corresponde à tabela de transições de estados. Esse diagrama ilustra claramente os eventos (intervalo de tempo) que disparam as mudanças de estado durante as transições. A ação em cada estado, que é manter a luz específica na saída, é representada nos retângulos. O diagrama foi desenhado com uso da ferramenta draw.io.



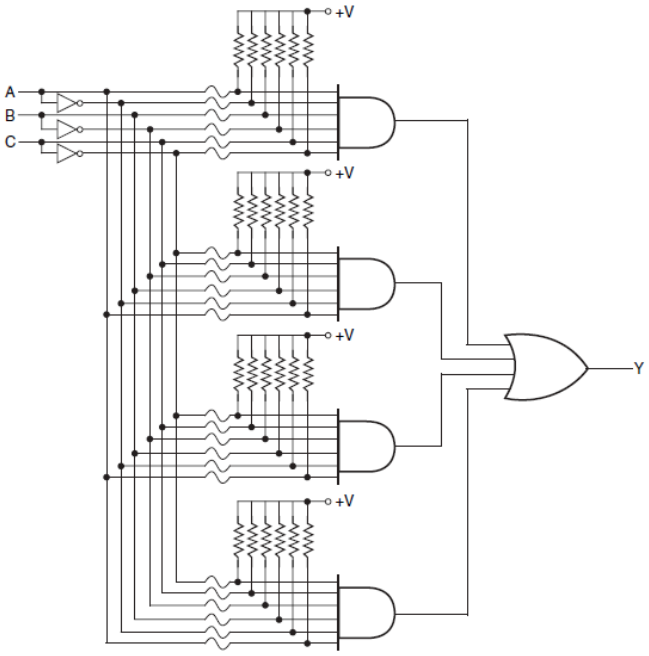
DISPOSITIVOS DE LÓGICA PROGRAMÁVEL

Tradicionalmente, o projeto de circuitos digitais dependia da análise manual de tabelas-verdade, mapas de Karnaugh ou do algoritmo de Quine-McCluskey, culminando no desenho de esquemáticos (que detalham as conexões físicas dos componentes, como ilustra a figura a seguir) e na implementação com componentes discretos. Nesse cenário, as máquinas de estados surgiram como uma ferramenta matemática para modelar e analisar o comportamento de sistemas digitais de forma mais estruturada. A formalização das Máquinas de Estados Finitas (FSMs) permitiu uma descrição abstrata desses sistemas, baseada em estados, transições e saídas. Essa abstração abriu o caminho para a síntese automática de circuitos, levantando a questão: seria possível, a partir de uma descrição comportamental abstrata, gerar automaticamente registradores, lógica combinacional e circuitos de controle?



Fonte: Geeksforgeeks

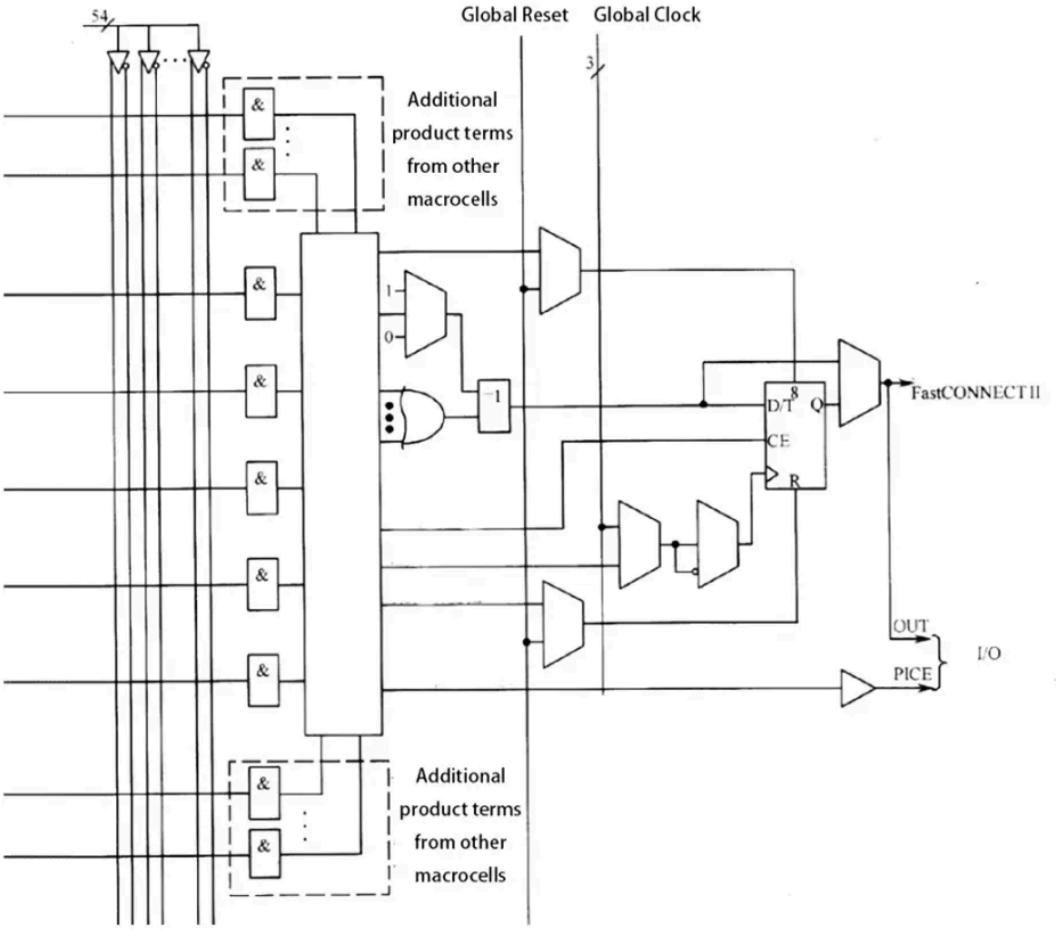
Foi nesse contexto que os **Dispositivos Lógicos Programáveis** (PLDs) ganharam destaque, oferecendo uma resposta a essa questão. Em vez de exigir a análise e otimização manual do comportamento de um circuito, os PLDs permitiram que os engenheiros descrevessem a lógica de forma abstrata, por exemplo, utilizando FSMs ou processos condicionais, e, então, deixassem que as ferramentas de síntese gerassem a implementação física diretamente no *chip*. A grande diferença em relação aos microprocessadores é que, enquanto a programação de um microprocessador altera os sinais de controle gerados por um *hardware* fixo, a programação de um PLD modifica as conexões internas entre suas portas lógicas. Essa dinâmica é bem ilustrada no diagrama lógico a seguir, que representa um PLD simples. A programação ocorre por meio de ligações fusíveis, ou "anti-fusíveis", que são projetadas para definir um mintermo específico na saída de cada porta AND. Inicialmente com alta resistência, esses anti-fusíveis são transformados em caminhos de baixa resistência quando submetidos a uma voltagem que excede um nível predeterminado.



Os componentes básicos de um dispositivo lógico programável mais complexo incluem:

- Blocos Lógicos Programáveis Simples, que são responsáveis pela implementação das funções lógicas desejadas.
- Matriz de Interconexão Programável, que permite a programação do roteamento dos sinais entre os blocos lógicos, facilitando a flexibilidade na configuração do circuito.
- Elementos de Memória, que armazenam estados e possibilitam a construção de circuitos sequenciais, que dependem do histórico das entradas.
- Circuitos de Entrada e Saída, que garantem a comunicação do dispositivo com o ambiente externo.

A figura a seguir ilustra esses componentes em um dispositivo lógico programável complexo (em inglês, *Complex Programmable Logic Device* – CPLD). No lado esquerdo, observa-se uma matriz de mintermos configurável, que permite a definição de diferentes operações lógicas combinacionais. À direita, um *flip-flop* programável pode ser configurado como *flip-flop* D ou T, possibilitando o armazenamento do estado do circuito conforme as necessidades do projeto. Se a saída deve depender exclusivamente das entradas, a saída da lógica combinacional pode ser direcionada diretamente para o pino de entrada/saída ou para a matriz de conexão.

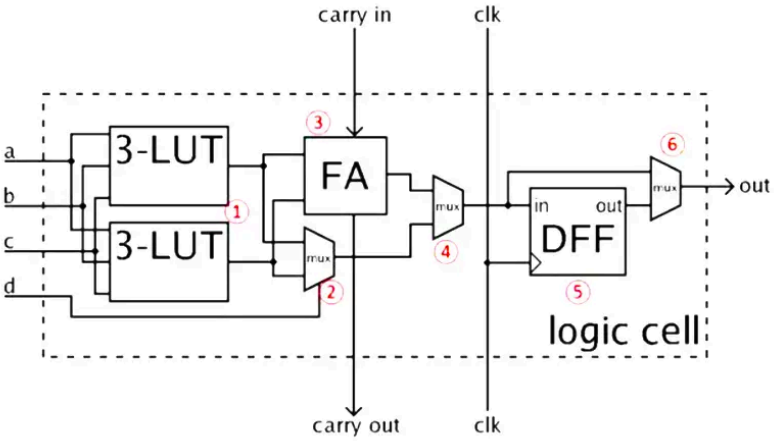


A evolução desses dispositivos, a partir do final da década de 1970, acompanha o progresso das necessidades de personalização e flexibilidade no desenvolvimento de *hardware*, bem como as demandas por maior capacidade e desempenho. A enciclopédia *online* Wikipedia oferece uma boa síntese sobre a evolução dessas tecnologias. Nos primórdios, em 1978, a MMI (*Monolithic Memories*) introduziu o **PAL** (do inglês *Programmable Array Logic*), um dispositivo que revolucionou a arquitetura dos PLDs existentes, tornando-os mais rápidos, compactos e econômicos. A popularidade dos PALs foi impulsionada pelo **PALASM** (do inglês *Programmable Array Logic Assembler*), uma linguagem de descrição de *hardware* (em inglês, *Hardware Description Language* – **HDL**) que permite a descrição das funções lógicas e interconexões, sendo sintetizável em padrões de fusíveis para programar esses dispositivos lógicos. Em 1985, a *Lattice Semiconductor* aprimorou o conceito com a **lógica de matriz genérica** (em inglês *Generic Array Logic* – **GAL**), que mantinha as mesmas propriedades lógicas dos PALs, mas permitia a reprogramação, facilitando a correção de erros de lógica durante a prototipagem. Em 1987, a introdução da linguagem de descrição de *hardware* **ABEL** (do inglês *A Better Environment for Logic*), com sua sintaxe clara e amigável para os desenvolvedores, contribuiu significativamente para a popularidade dos GALs. Tanto os PALs quanto os GALs, classificados como **SPLDs** (do inglês, *Simple Programmable Logic Devices*), eram limitados em tamanho, abrangendo apenas algumas centenas de portas lógicas. No final da década de 1980, foi introduzida a linguagem de descrição **CUPL** (do inglês *Common Universal Programmable Logic*), projetada para sintetizar descrições de *hardware* baseadas em tabelas para as tecnologias PALs e GALs.

A progressão continuou com os **dispositivos lógicos programáveis complexos** (em inglês *Complex Programmable Logic Devices* – **CPLDs**), que combinavam a lógica de vários PALs em um único circuito integrado, permitindo a implementação de milhares ou até centenas de milhares de portas lógicas. Simultaneamente, surgiram as matrizes de porta programáveis (em inglês, *Field-Programmable Gate Arrays* – **FPGAs**), que utilizavam uma matriz de portas lógicas configurável pelo usuário. A principal diferença entre CPLDs e FPGAs reside na arquitetura interna de cada bloco lógico: os CPLDs empregam uma estrutura de "*sea-of-gates*", enquanto as FPGAs utilizam LUTs (do inglês *Look-Up Tables*).

A estrutura "*sea-of-gates*" consiste em uma matriz de portas lógicas interconectadas. Nessa configuração, diferentes tipos de portas, como AND, OR e NOT, podem ser distribuídas ao longo da matriz, permitindo uma flexibilidade significativa na implementação de funções lógicas. A interconexão entre essas portas pode ser modificada durante a programação, o que facilita **a criação de circuitos lógicos combinacionais**. Essa estrutura é especialmente adequada para aplicações que requerem uma lógica relativamente simples e direta. Por outro lado, as LUTs, (do inglês *LookUp Table*) presentes nas **FPGAs** oferecem uma abordagem distinta para a lógica digital. Cada LUT atua como uma pequena tabela de busca que armazena os resultados de uma função lógica para todas as combinações possíveis de entradas. Durante a programação da FPGA, essas LUTs são configuradas para implementar diversas funções lógicas, permitindo que uma única LUT realize múltiplas operações. Além disso, várias LUTs podem operar simultaneamente, proporcionando um alto nível de paralelismo, o que torna as FPGAs ideais para **implementação de máquinas de estados complexas**, como microprocessadores.

A figura a seguir mostra uma versão simplificada de um bloco lógico configurável de uma FPGA que consiste em duas LUTs de três entradas (1), um somador completo (3), um *flip-flop* tipo D (5), um multiplexador regular (2) e dois multiplexadores programáveis (4) e (6).



Fonte: 6G Controls

Para a síntese de tecnologias complexas, como CPLDs e FPGAs, foi introduzida em 1981 a linguagem de descrição **VHDL** (do inglês *VHSIC Hardware Description Language*), inicialmente desenvolvida pelo Departamento de Defesa dos EUA como parte do projeto VHSIC (do inglês *Very High Speed Integrated Circuit*). Essa linguagem foi projetada para descrever o comportamento e a estrutura de circuitos digitais, facilitando a documentação e a simulação de sistemas eletrônicos. Em 1984, surgiu o **Verilog**, desenvolvido pela empresa privada *Gateway Design Automation*, com um foco mais comercial, buscando se adaptar rapidamente às necessidades da indústria. Verilog introduziu conceitos de modelagem que simplificaram a simulação de circuitos digitais. Sua abordagem à descrição de comportamento e à modelagem de tempo contribuiu para sua popularidade entre engenheiros que precisavam de ferramentas eficientes. Com o tempo, Verilog se estabeleceu como uma das principais linguagens de descrição de *hardware*, competindo diretamente com VHDL. Algumas empresas e setores preferiram Verilog devido à sua simplicidade e eficiência. No entanto, é importante ressaltar que Verilog não foi desenvolvido como uma alternativa direta ao VHDL, mas sim como uma solução que atendia a diferentes requisitos e preferências dentro da comunidade de *design* eletrônico. Ambas as linguagens são padronizadas pelo IEEE.

Em um nível mais avançado, os **circuitos integrados de aplicação específica** (em inglês, *Application Specific Integrated Circuits* – **ASICs**) representam um passo significativo na especialização da lógica programável. Diferente das FPGAs e CPLDs, que podem ser programados e reprogramados pelo usuário após a fabricação, os ASICs são projetados e fabricados para uma aplicação específica. A configuração de um ASIC é definida durante as etapas de *design* e fabricação, onde a lógica e a topologia do circuito são cuidadosamente estabelecidas. Embora os ASICs não possam ser reprogramados após a produção, seu *design* inicial é realizado com o uso de linguagens de descrição de *hardware*, permitindo que os engenheiros especifiquem a lógica e as interconexões necessárias para a aplicação. Esse processo resulta em um *chip* otimizado para executar exclusivamente a função para a qual foi projetado, oferecendo alto desempenho, baixo consumo de energia e eficiência máxima para sua aplicação específica.

O uso de dispositivos de lógica programável revolucionou o desenvolvimento de *hardware* digital. Sua capacidade de reconfiguração após a fabricação trouxe benefícios significativos, como maior flexibilidade, adaptação rápida a novas especificações, prototipagem acelerada e otimização de desempenho para soluções personalizadas. A evolução tecnológica dos PALs até os ASICs impulsionou a inovação na eletrônica, ampliando as possibilidades para atender às crescentes demandas por desempenho, eficiência e versatilidade. No entanto, a escolha da tecnologia ideal para cada aplicação depende de uma análise criteriosa, levando em conta fatores como custo, tempo de desenvolvimento, consumo de energia e volume de produção.

Para configurar esses dispositivos de acordo com o comportamento desejado, é necessário especificar esse comportamento de forma precisa, permitindo que ferramentas de síntese gerem os circuitos apropriados. Atualmente, existem duas abordagens complementares para essa especificação:

- As linguagens de descrição de *hardware* (em inglês, *Hardware Description Languages* – HDLs), como VHDL e Verilog, são linguagens textuais, ideais para descrever o que o circuito deve fazer, permitindo trabalhar em um nível mais abstrato e comportamental.

- As capturas esquemáticas (em inglês, *schematic captures*), por outro lado, são úteis para representar visualmente como os componentes estão conectados, sendo mais intuitivos em projetos menores ou na análise estrutural de circuitos.

Ambas as abordagens são fundamentais no processo de desenvolvimento moderno, e muitas vezes são utilizadas em conjunto, combinando clareza estrutural com precisão funcional.

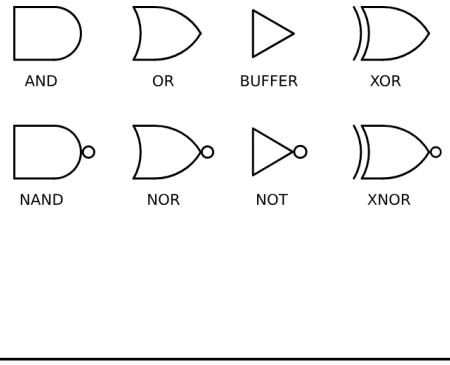
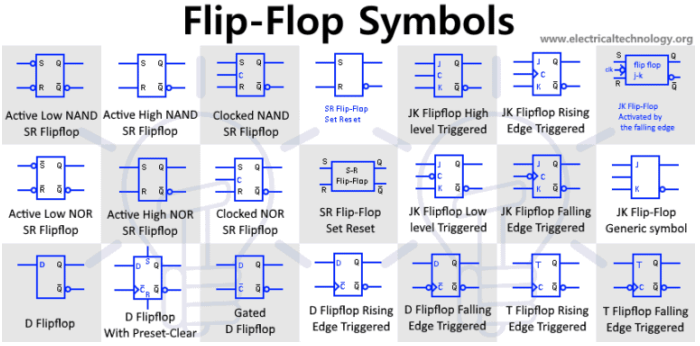
Captura esquemática

Os esquemáticos possuem um papel fundamental e de longa data no *design* de circuitos digitais. Inicialmente, os circuitos eram desenhados à mão em papel, utilizando símbolos para representar portas lógicas, *flip-flops* e outros componentes. Com a digitalização dessa prática, surgiram as ferramentas de **captura esquemática**. Antes da dominância das HDLs, como VHDL e Verilog, os esquemáticos eram a principal forma de descrever a estrutura de um circuito digital. Engenheiros interligavam visualmente os blocos lógicos, simulavam seu comportamento e, por fim, implementavam o circuito em placas de circuito impresso (PCBs) ou ASICs. A grande familiaridade dos engenheiros com essa representação visual é, portanto, um legado histórico.

Um **esquemático** é uma representação gráfica de um circuito eletrônico, composta por elementos básicos que simbolizam suas funcionalidades:

- Símbolos de Componentes: Representações padronizadas de portas lógicas (AND, OR, NOT), *flip-flops* (D, JK, T), multiplexadores, registradores, contadores, e outros blocos funcionais. Cada símbolo indica a função e as portas de conexão do componente.
- Conexões (em inglês, *Nets*/*Wires*): Linhas que interligam as portas dos componentes, indicando a rota de sinais elétricos.
- Rótulos e Nomes: Identificadores para sinais, barramentos e componentes específicos, facilitando a leitura e a referência cruzada.
- Barramentos: Linhas mais grossas ou agrupamentos de linhas que representam múltiplos sinais relacionados, como um barramento de dados de 8 *bits*.

Símbolos em capturas esquemáticas	
Portas lógicas	*Flip-flops*

	
Fonte: SpinningNumbers	Fonte: Electrical Technology

Com o advento dos Dispositivos Lógicos Programáveis (PLDs) e, posteriormente, das FPGAs, a capacidade de programar o *hardware* abriu novas fronteiras. Embora as HDLs se tornassem a ferramenta preferencial para descrever comportamentos complexos, como máquinas de estados, que seriam inviáveis de desenhar em um esquemático completo, as ferramentas de desenvolvimento para PLDs/FPGAs, como o Quartus Prime, mantiveram o suporte à entrada esquemática. Atualmente, os esquemáticos continuam sendo relevantes. Eles são particularmente úteis para representar sub-blocos de *designs* ou para a interconexão visual de módulos maiores, oferecendo uma clareza visual superior. É comum em um *design* complexo que a maior parte da lógica seja escrita em VHDL/Verilog, mas os blocos principais sejam conectados visualmente em um esquemático de nível superior. Além disso, em contextos específicos, como com componentes legados ou em projetos onde a visualização da interconexão física é crucial, os esquemáticos permanecem a representação padrão.

Linguagem de descrição de *hardware*

As Linguagens de Descrição de *Hardware* (em inglês *Hardware Description Languages* – HDLs), como VHDL e Verilog, são linguagens de configuração criadas especificamente para descrever o comportamento e a estrutura de circuitos digitais. Diferente de linguagens tradicionais de *software*, as HDLs permitem representar com precisão como o *hardware* deve funcionar e como seus componentes estão interconectados. As HDLs permitem que projetistas descrevam circuitos digitais em diferentes níveis de abstração, desde a definição de portas lógicas básicas, registradores, contadores, até estruturas complexas como controladores de protocolos e processadores. Em VHDL, as estruturas fundamentais envolvem:

- Entidades (em inglês, *entity*) e Arquiteturas (em inglês, *architecture*): separam a interface (entradas e saídas) da descrição interna do circuito.
- Sinais (em inglês, *signal*) e variáveis (em inglês, *variable*) : armazenam e transmitem sinais/valores dentro do circuito.
- Processos (em inglês, *process*): descrevem blocos de comportamento síncrono ou assíncrono.
- Atribuições condicionais e sequenciais: modelam a lógica do circuito.

- Temporização e sensibilidade ao *clock*: essencial para circuitos sequenciais.

A seguir, apresentamos dois exemplos de descrição de *hardware*. O código à esquerda detalha um circuito combinacional que implementa a lógica $\neg(A \wedge B)$ (NAND), enquanto o código à direita descreve o *flip-flop* D com *reset* assíncrono, um componente básico de um circuito sequencial, encapsulado em um bloco `process`. Observe a presença das estruturas básicas nas duas descrições. Os símbolos desses dois componentes usados em esquemáticos são mostrados na figura anterior.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity circuito_simples is
    Port (
        A : in  std_logic;
        B : in  std_logic;
        Y : out std_logic
    );
end circuito_simples;

architecture comportamento of
circuito_simples is
    signal Q : std_logic;  -- sinal
intermediário (saída da AND)
begin

    Q <= A AND B;  -- Porta AND
    Y <= NOT Q;    -- Porta NOT

end comportamento;
```

```
library ieee;
use ieee.std_logic_1164.all;

entity D_FlipFlop_AsyncReset is
    Port (
        clk   : in  std_logic;
        reset : in  std_logic; -- Reset assíncrono (ativo alto)
        D_in  : in  std_logic;
        Q_out : out std_logic
    );
end entity D_FlipFlop_AsyncReset;

architecture comportamento of D_FlipFlop_AsyncReset is
begin
    -- Este é o bloco process
    process (clk, reset) -- Lista de sensibilidade: o processo reage a
mudanças em clk OU reset
    begin
        -- Lógica de reset assíncrono: se reset for '1', Q_out vai para
'0' imediatamente
        if reset = '1' then
            Q_out <= '0';
        -- Lógica síncrona: se reset não está ativo, reage à borda de
subida do clock
        elsif rising_edge(clk) then
            Q_out <= D_in; -- Na borda de subida do clock, Q_out
recebe o valor de D_in
        end if;
    end process;

end comportamento;
```

HDLs podem ser utilizadas em diversas etapas do desenvolvimento de sistemas digitais. Elas permitem a **simulação** do comportamento funcional de um circuito antes mesmo de sua implementação física, garantindo a sua correção. Em seguida, na fase de **síntese**, o código HDL é convertido em uma implementação física, seja uma malha de portas lógicas ou um circuito configurável em uma FPGA. Por fim, a **verificação** assegura que o circuito descrito atenda tanto aos requisitos funcionais quanto aos temporais. Esse fluxo é indispensável para os projetos modernos, especialmente com o aumento contínuo da complexidade dos sistemas digitais.

Uma das aplicações mais comuns de HDLs está na descrição de lógica sequencial, onde o comportamento do circuito depende de eventos passados. Esse tipo de lógica é naturalmente

modelado por Máquinas de Estados Finitos (FSMs). Em VHDL, por exemplo, FSMs são amplamente usadas para descrever controladores de estado, protocolos de comunicação, sequenciadores de tarefas e outros circuitos que evoluem ao longo do tempo com base em entradas e estados anteriores. Assim, compreender bem o conceito de FSMs é essencial para escrever código VHDL mais organizado e legível, garantindo transições de estado corretas e estruturando as saídas com clareza, seja no modelo de Moore ou Mealy.

VHDL

Embora existam diversas linguagens de descrição de *hardware*, as mais utilizadas são VHDL e Verilog. Ambas são amplamente adotadas na indústria e na academia, mas possuem características distintas que as tornam mais adequadas a diferentes contextos de projeto. Alguns desenvolvedores preferem VHDL por sua estrutura sintática mais rigorosa e sistema de tipagem forte, que exige a definição explícita dos tipos de dados e ajuda a prevenir erros comuns de codificação. Essas características são especialmente valiosas em projetos complexos, que exigem alta confiabilidade, documentação clara e facilidade de manutenção ao longo do tempo. Além disso, VHDL é amplamente adotado em ambientes acadêmicos e setores que valorizam documentação e padronização formal, o que contribui para uma rica base de materiais educacionais e técnicos disponíveis. Por outro lado, Verilog é frequentemente escolhido por sua sintaxe mais simples e curva de aprendizado mais suave, sendo muito utilizado em projetos de menor complexidade ou em ambientes onde o desenvolvimento rápido é uma prioridade. Neste roteiro, optamos por utilizar VHDL como linguagem principal para a descrição dos projetos de *hardware*, e a seguir apresentamos uma visão geral dos principais elementos da sua sintaxe.

O autor P. P. Chu apresenta a figura que se segue para ilustrar a abrangência e a cobertura do VHDL em um fluxo de desenvolvimento simplificado para projetos de *hardware*. O *design* de um sistema complexo geralmente começa com uma descrição abstrata em alto nível, que define o comportamento desejado do sistema, acompanhada de um banco de testes (em inglês, *testbench*) que contém um conjunto de vetores de teste para avaliar diversas funções. Essa combinação de descrição e banco de testes permite que os projetistas analisem detalhadamente a operação do sistema, identifiquem eventuais equívocos ou inconsistências, refinem e finalizem as especificações e, finalmente, estabeleçam o comportamento desejado de entrada e saída para futuras verificações.

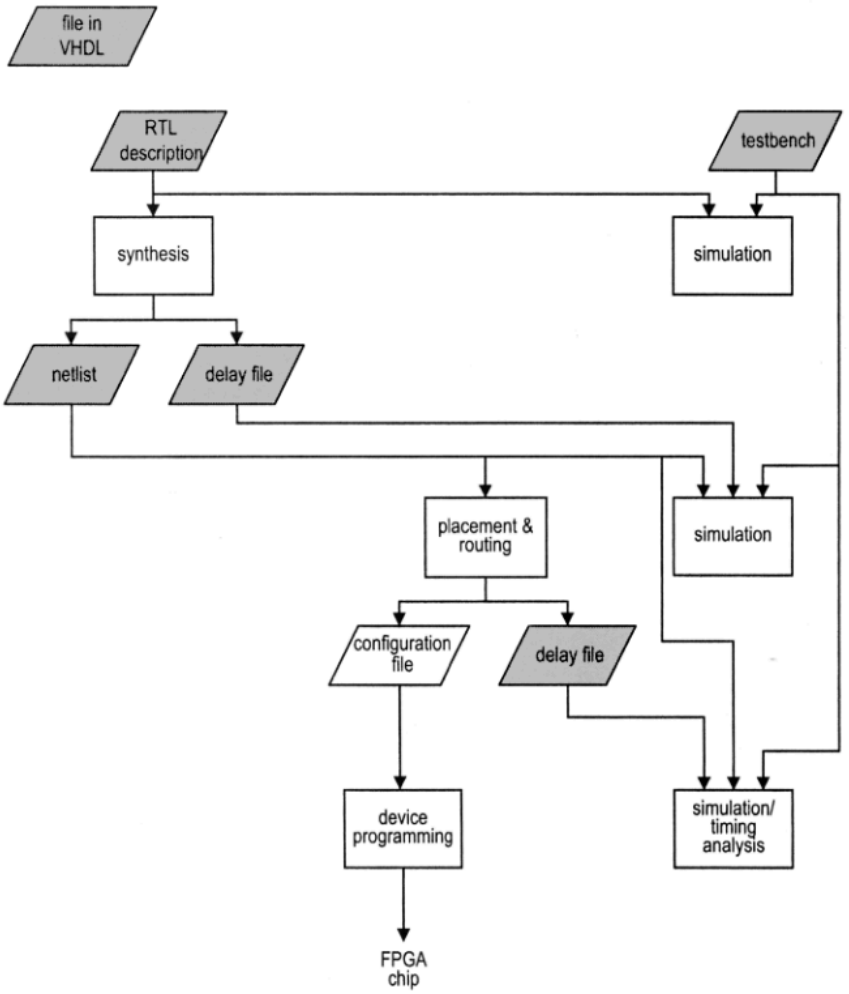


Figure 2.8 Coverage of VHDL in development flow.

Estrutura básica

Um arquivo VHDL sintetizável em implementações físicas precisa de pelo menos duas partes principais: uma declaração de entidade e um corpo de arquitetura associado à entidade. A declaração de **entidade** define a interface da entidade com o ambiente externo, especificando o nome da entidade (*entity_name*), os nomes de suas portas (*port_names*), o sentido do fluxo de sinal (*mode*) e o tipo de dados associado a cada porta (*port*). Os modos de fluxo de sinal incluem: entrada (`in`), saída (`out`) e entrada e saída (`inout`). O formato básico de uma declaração de entidade é:

```
entity entity_name is
   port(
      port_names: mode data_type;
      port_names: mode data_type;
      ...
      port_names: mode data_type
   );
end entity_name;
```

O corpo da **arquitetura** especifica a operação interna ou a organização de um circuito. Em VHDL, podemos desenvolver múltiplos corpos de arquitetura para a mesma declaração de entidade e, posteriormente, escolher um corpo para vincular à entidade para simulação ou síntese. O formato básico de definição de um corpo de arquitetura é:

```
architecture arch_name of entity_name is
   declarations;
begin
   concurrent statement;
   concurrent statement;
   concurrent statement;
   . . .
end arch_name;
```

Na **primeira linha**, especifica-se o nome da entidade (*entity_name*) associada e o nome (*arch_name*) que se deseja atribuir à arquitetura. Embora o nome da arquitetura em VHDL seja arbitrário, seguir algumas melhores práticas pode facilitar a compreensão e a manutenção do código. É recomendável que o nome da arquitetura reflita seu propósito ou tipo de implementação. Por exemplo, utilizar nomes como comportamental, estrutural, RTL (do inglês *Register Transfer Level*) ou FSM (do inglês *Finite State Machine*) pode ajudar a indicar a abordagem utilizada. Manter uma convenção de nomenclatura consistente ao longo de todo o projeto é fundamental, pois isso permite identificar rapidamente o tipo de arquitetura aplicada em diferentes entidades. Além disso, evite abreviações ou siglas que possam não ser claras para outros desenvolvedores, pois um nome descritivo promove melhor colaboração e legibilidade do código. Caso existam múltiplas arquiteturas para uma mesma entidade (como diferentes modos de operação), considere incluir um sufixo ou prefixo que indique a variante.

As **linhas** que seguem são reservadas para a declaração dos sinais internos da entidade. É comum que declarações de sinais sejam distribuídas em várias linhas para melhorar a legibilidade e a organização do código. Em seguida, é apresentada uma **descrição detalhada** da funcionalidade da entidade através de declarações concorrentes (*concurrent statement*). Esta descrição é **delimitada pelas palavras-chave** `begin` **e** `end`. A **natureza concorrente** é uma característica distintiva da descrição de *hardware*. Diferentemente da execução sequencial de instruções em linguagens de programação tradicionais, as instruções em VHDL são independentes e podem ser ativadas e executadas em paralelo, uma vez que cada instrução é sintetizada como uma parte distinta do circuito.

Tipos de dados

Existe cerca de uma dúzia de tipos de dados pré-definidos. Apenas os seguintes tipos de dados são frequentemente utilizados para síntese:

- `integer`: O VHDL não define o intervalo exato do tipo inteiro, mas especifica que o intervalo mínimo é de -(2^{31} - 1) a 2^{31} - 1, correspondendo a 32 *bits*. Dois tipos de dados relacionados (formalmente conhecidos como subtipos) são os tipos `natural` e `positive`. O primeiro inclui 0 e números positivos, enquanto o segundo inclui apenas números positivos.
- `boolean`: Definido como (false, true).
- `bit`: Definido como ("0", "1").
- `bit_vector`: Definido como um *array* unidimensional com elementos do tipo `bit`. A intenção original do tipo `bit` é representar os dois valores binários usados na álgebra booleana e na lógica digital. No entanto, em um *design* real, um sinal pode assumir outros valores, como a alta impedância da saída de um *buffer* de três estados ou um valor "conflitante" devido a um conflito (por exemplo, duas saídas conectadas, formando um curto-circuito). Para resolver esse problema, um conjunto de tipos de dados mais versáteis, **`std_logic`** e **`std_logic_vector`**, foi introduzido no pacote IEEE `std_logic_1164`. **Para melhor compatibilidade, deve-se evitar o uso dos tipos `bit` e `bit_vector`**.

Operadores

Cerca de 30 operadores são definidos na VHDL. Em uma linguagem fortemente tipada, a definição de um tipo de dado inclui as operações que podem ser realizadas sobre objetos desse tipo. É importante saber quais tipos de dados podem ser usados com um operador específico. A precedência e as descrições desses operadores em VHDL-93, bem como os tipos de dados, relevantes para a síntese em *hardware*, estão resumidas na tabela de precedência e tabela de operadores a seguir.

Table 3.2 Precedence of the VHDL operators

Precedence	Operators
Highest	** abs not
	* / mod rem
	+ - (identity and negation)
	& + - (addition and subtraction)
	sll srl sla sra rol ror
	= /= < <= > >=
Lowest	and or nand nor xor xnor

Operator	Description	Data type of operand a	Data type of operand b	Data type of result
a ** b	exponentiation	integer	integer	integer
abs a	absolute value	integer		integer
not a	negation	boolean, bit, bit_vector		boolean, bit, bit_vector
a * b	multiplication	integer	integer	integer
a / b	division			
a mod b	modulo			
a rem b	remainder			
+ a	identity	integer		integer
- a	negation			
a + b	addition	integer	integer	integer
a - b	subtraction			
a & b	concatenation	1-D array, element	1-D array, element	1-D array
a sll b	shift-left logical	bit_vector	integer	bit_vector
a srl b	shift-right logical			
a sla b	shift-left arithmetic			
a sra b	shift-right arithmetic			
a rol b	rotate left			
a ror b	rotate right			
a = b	equal to	any	same as a	boolean
a /= b	not equal to			
a < b	less than	scalar or 1-D array	same as a	boolean
a <= b	less than or equal to			
a > b	greater than			
a >= b	greater than or equal to			
a and b	and	boolean, bit, bit_vector	same as a	same as a
a or b	or			
a xor b	xor			
a nand b	nand			
a nor b	nor			
a xnor b	xnor			

Por fim, vale mencionar algumas notações muito usadas na VHDL. O operador de atribuição de sinais é representado por "<=", indicando uma alteração "não imediata" do sinal do lado esquerdo do operador. **É importante notar que uma porta de saída não pode aparecer no lado direito de uma instrução de atribuição de sinais, pois isso violaria a lógica do fluxo de dados.** A notação `=>` é utilizada principalmente para associar valores a parâmetros em chamadas de subprogramas, como funções e procedimentos, e para conectar sinais em instruções de mapeamento ao instanciar componentes. Quando uma função ou procedimento é chamado, pode-se usar `=>` para especificar o nome do parâmetro e o valor a ser atribuído,

como em `my_function(param1 => value1, param2 => value2)`. Além disso, ao instanciar um componente, `=>` é empregado para mapear as portas do componente às linhas correspondentes doprojeto, por exemplo, `my_component: my_entity port map (signal_a => input_signal, signal_b => output_signal)`. Já o operador "**:=**" é utilizado para atribuir "imediatamente" um valor, seguindo a convenção de atribuição de valores em linguagem de programação tradicional. Em descrições VHDL, tudo o que aparece após "**--**" na mesma linha é tratado como um comentário e não é processado pelo compilador. Isso permite que os desenvolvedores incluam observações e esclarecimentos no código sem afetar sua execução.

Instruções concorrentes

De acordo com a definição da VHDL, a instrução de atribuição de sinal concorrente possui duas formas básicas: a instrução de atribuição condicional de sinal e a instrução de atribuição selecionada de sinal. Uma instrução de atribuição condicional simples ao sinal `signal_name` pode ser escrita como

```
signal_name <= projected_waveform;
```

onde `projected-waveform` consiste em dois tipos de especificações: a expressão de um novo valor para o sinal e o tempo em que o novo valor ocorrerá

```
signal_name <= value_expression after timing;
```

onde `value_expression` pode ser um valor constante, operação lógica, operação aritmética, e assim por diante. O aspecto temporal de `projected_waveform` normalmente corresponde ao atraso de propagação interno para completar o cálculo da expressão. No entanto, como o atraso de propagação depende dos componentes, da tecnologia do dispositivo, do roteamento, do processo de fabricação e do ambiente de operação, é impossível sintetizar um circuito com uma quantidade exata de atraso. **Portanto, para síntese, informações explícitas de tempo não são especificadas no código VHDL**. O valor de atraso padrão δ-atraso é aplicado no sinal projetado implicitamente e a instrução assume o seguinte aspecto

```
signal_name <= value_expression;
```

Esta instrução pode ser pensada como um bloco de circuito, onde a saída do circuito é o sinal no lado esquerdo da instrução, e as entradas são todos os sinais que aparecem em `value_expression` do lado direito.

Uma instrução de atribuição condicional pode envolver várias expressões booleanas. Essas expressões booleanas são avaliadas sucessivamente até que uma seja considerada verdadeira, e a expressão de valor correspondente seja atribuída ao sinal de saída. Em outras palavras, a primeira expressão booleana, `boolean_expr_1`, é verificada primeiro. Se for verdadeira, a

primeira expressão de valor, `value_expr_1`, será atribuída ao sinal de saída. Se for falsa, a segunda expressão booleana, `boolean_expr_2`, será verificada a seguir. Esse processo continua até que todas as expressões booleanas sejam verificadas. A última expressão de valor, `value_expr_n`, será atribuída ao sinal caso nenhuma das expressões booleanas seja verdadeira.

```
signal_name <= value_expr_1 when boolean_expr_1 else
               value_expr_2 when boolean_expr_2 else
               value_expr_3 when boolean_expr_3 else
                  . . .
               value_expr_n;
```

A síntese de uma instrução de atribuição de sinal condicional requer três grupos de *hardware*: circuitos de `value_expr`, circuitos de `boolean_expr` e rede de roteamento por prioridade. Os circuitos de `value_expr` realizam as expressões de valor, `value_expr_1`, …, `value_expr_n`, e um dos resultados é direcionado para a saída. Os circuitos de `boolean_expr` realizam as expressões booleanas, `boolean_expr_1`, ..., `boolean_expr_(n-1)`, e seus valores são usados para controlar a rede de roteamento por prioridade. A rede de roteamento por prioridade é a estrutura que roteia e controla o valor desejado para o sinal de saída. Embora a construção seja direta, é difícil para um sintetizador transformar uma rede de circuitos de `boolean_expr` extremamente profunda em uma implementação eficiente. Assim, devemos considerar o impacto no número de palavra reservada **when** ao escrever uma atribuição de sinal condicional.

Uma instrução de atribuição selecionada de sinal representa um circuito abstrato de multiplexação, no qual o sinal `select_expression` determina qual dos resultados `value_expr_1`, ..., `value_expr_n` será direcionado à saída, com base no valor de `select_expression`, que escolhe entre as opções `choice_1`, ..., `choice_n`. Neste circuito de multiplexação, cada valor possível de `select_expression` tem uma porta de entrada designada no multiplexador, e `select_expression` funciona como o sinal de seleção deste multiplexador. Uma vez que seu valor é determinado, o resultado da expressão de valor designada é passado para a porta de saída do multiplexador.

```
with select_expression select
   signal_name <= value_expr_1 when choice_1,
                  value_expr_2 when choice_2,
                  value_expr_3 when choice_3,
                     . . .
                  value_expr_n when choice_n;
```

Todas as instruções de atribuição de sinal selecionado possuem uma síntese semelhante. A principal diferença está no número de valores que a expressão de seleção pode assumir, o

que, por sua vez, determina o tamanho do multiplexador. Apesar da construção conceitual simples, certas tecnologias de dispositivos podem ter dificuldade em suportar um circuito de multiplexação extremamente amplo. Portanto, é importante levar em conta o número de valores em uma expressão de seleção na descrição de uma atribuição selecionada de sinal.

Instruções sequenciais

As declarações sequenciais em VHDL são executadas em uma ordem definida, de forma semelhante às instruções em linguagens de programação tradicionais. No entanto, elas só podem ser utilizadas dentro de um bloco `process`, que, por sua vez, é uma estrutura concorrente. O principal propósito das declarações sequenciais é descrever o "comportamento abstrato" do circuito, permitindo modelar seu funcionamento lógico sem precisar representar diretamente os componentes físicos envolvidos. Isso contrasta com as instruções concorrentes, como as atribuições com o operador `<=` fora de `process`es, que tendem a refletir de forma mais direta a estrutura física do *hardware* sintetizado.

Para que um código VHDL seja corretamente sintetizado, é fundamental que ele seja escrito de forma organizada, estruturada e aderente às boas práticas de modelagem. Um projeto bem escrito garante que o comportamento descrito seja interpretado de forma clara e precisa pelas ferramentas de síntese, resultando em uma implementação de *hardware* fiel, eficiente e confiável. Um dos aspectos mais importantes nesse processo é a forma como os blocos lógicos são segmentados dentro da descrição. Misturar, dentro de um mesmo bloco `process`, lógica combinacional e sequencial pode gerar ambiguidade ou dificultar a interpretação do comportamento esperado pelo sintetizador. Embora o código possa ser funcionalmente correto e simular conforme o esperado, essa abordagem tende a prejudicar a otimização durante a síntese física do circuito.

Os sintetizadores atuais ainda apresentam limitações para inferir corretamente os limites entre lógica sequencial (baseada em *flip-flops*, controlada por *clock*) e lógica puramente combinacional quando ambas estão misturadas em um mesmo processo. Isso pode resultar em circuitos com lógica desnecessária, retardos maiores e uso ineficiente de recursos, como LUTs e *flip-flops* nos FPGAs. Para evitar esses problemas, a prática recomendada é separar claramente os blocos lógicos numa estratégia conhecida como "*n-Process FSM*":

- Um bloco `process` exclusivamente sequencial, sensível apenas ao *clock* e *reset*, responsável por armazenar o estado interno (registradores, contadores, etc.).
- Um ou mais (n-1) blocos `process` puramente combinacionais, geralmente usados para gerar as saídas ou calcular o próximo estado, sensíveis apenas aos sinais de entrada e/ou ao estado atual.

A seguir, apresenta-se o formato típico de declaração de um `process`, incluindo sua lista de sensibilidade (`sensitivity_list`), que especifica os sinais aos quais o processo deve reagir.

```
process(sensitivity_list)
   declarations;
begin
   sequential statement;
   sequential statement;
  . . .
end process;
```

As instruções sequenciais ainda incluem:

wait: suspende a execução de um processo até que uma condição específica seja atendida; durante o tempo em que o processo está suspenso, ele não executa nenhuma ação, mas aguarda o evento que ativa o `process`.

```
wait on signals;
wait until boolean_expression;
wait for time_expression;
```

atribuição não-imediata de sinal sequencial: é idêntica à da atribuição de sinal concorrente simples, exceto pelo fato que a instrução está dentro de um `process`.

if: realiza decisões condicionais dentro de um `process`.

```
if boolean_expr_1 then
   sequential_statements;
elsif boolean_expr_2 then
   sequential_statements;
elsif boolean_expr_3 then
   sequential_statements;
. . .
else
   sequential_statements;
end if;
```

when … else: define o valor de saída com `value_expr_i` se (**when**) o valor da expressão booleana `boolean-expr-i` é verdadeira, senão (**else**) testa-se a próxima condição.

```
sig <= value_expr_1 when boolean_expr_1 else
       value_expr_2 when boolean_expr_2 else
       value_expr_3 when boolean_expr_3 else
          . . .
       value_expr_n;
```

case: realiza uma seleção condicional, permitindo que o código execute diferentes blocos de instruções com base no valor de uma expressão (geralmente uma variável ou sinal).

```
case case_expression is
   when choice_1 =>
      sequential statements;
   when choice_2 =>
      sequential statements;
   . . .
   when choice_n =>
      sequential statements;
end case;
```

for … in … loop: executa um bloco de código repetidamente, um número fixo de vezes, de acordo com a definição de um intervalo (ou intervalo de índices).

```
for index in loop_range loop
    sequential statements;
end loop;
```

Pacotes e bibliotecas

Um pacote VHDL geralmente abriga uma coleção de itens frequentemente utilizados, como tipos de dados, subprogramas e componentes, que podem ser reutilizados em diversos programas VHDL. Para facilitar a síntese, a IEEE desenvolveu vários pacotes VHDL, incluindo os pacotes `std_logic_1164` e `numeric_std`, conforme definido nas normas IEEE 1164 e 1076.3. Para utilizar um pacote pré-definido, é necessário declarar a biblioteca que o contém com a palavra reservada `library`, seguida da instrução `use` para tornar o pacote acessível, tudo isso antes da declaração da entidade.

Banco de testes

VHDL é uma linguagem usada para descrever circuitos digitais, mas vimos que nem todas as instruções que escrevemos nela podem ser transformadas diretamente em circuitos físicos. Ou seja, existem comandos e estruturas que são usados apenas para descrever a lógica de organização dos circuitos físicos e não para construção do *hardware* real. Portanto, é comum durante o processo de desenvolvimento, faz-se várias simulações para testar a descrição de *hardware* até que ele coincida com as especificações do projeto e sintetizável. Um *testbench* é um tipo de arquitetura usado para testar e validar um módulo. Normalmente, ele não é feito para ser sintetizado (ou seja, transformado em *hardware* físico). Portanto, um *testbench* geralmente está associado a uma entidade sem funcionalidade externa, não necessitando de interfaces com o mundo exterior. A forma mais comum de construir um *testbench* é por meio do modelo DUT (do inglês, *Device Under Test*) ou UUT (don inglês, *Unit Under Test*). Nesse modelo, são aplicados sinais de entrada, conhecidos como **estímulos de teste**, para simular o comportamento do circuito. Isso permite verificar se o projeto está funcionando corretamente antes de avançar para a fase de síntese, na qual ele será convertido em um circuito físico real.

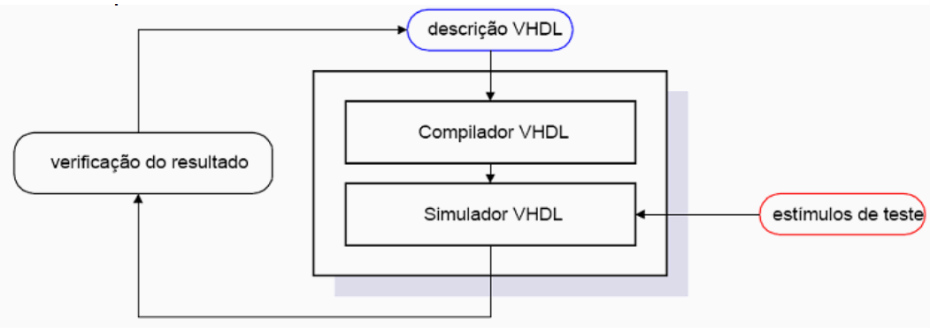


Figura 2.3 Etapa de elaboração da descrição VHDL.

Os principais elementos de um *testbench* são

- `entity`: Sempre vazia em *testbenches*.
- `architecture`: Contém sinais, DUT, e processos de estímulo.
- `component`: Declara o circuito a ser testado.
- `signal`: Declara os sinais usados na simulação.
- `process`: Define o clock e os estímulos.
- `wait`: Controla o tempo de simulação.

Exemplos

Vamos ilustrar a descrição VHDL de dois projetos: um contador bidirecional de 4 bits e o circuito de controle de um semáforo descrito em UML (Seção UML).

A descrição do contador bidirecional `UpDownCounter` de 4 *bits* inclui os pacotes STD_LOGIC_1164, NUMERIC_STD e STD_LOGIC_UNSIGNED da biblioteca IEEE, permitindo o uso dos tipos de dados básicos e lógicos desenvolvidos pela IEEE e a realização de operações aritméticas com e sem sinal, todas **sintetizáveis em circuitos de dispositivos lógicos programáveis**:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

A seguir, apresentamos a declaração da entidade `UpDownCounter`, que possui quatro sinais digitais do tipo `STD_LOGIC` e um vetor de sinais digitais do tipo `STD_LOGIC_VECTOR`, fundamentais para o funcionamento de um contador bidirecional de 4 *bits*.

```
entity UpDownCounter is
      Port (
            clk    : in  STD_LOGIC; - sinal de clock
            rst    : in  STD_LOGIC;  -- Reset síncrono
            enable: in  STD_LOGIC;  -- Enable para contagem
            up_down: in  STD_LOGIC; -- Sinal para contagem crescente (1) ou
  decrescente (0)
            count: out STD_LOGIC_VECTOR(3 downto 0)   -- Saída do contador
  de 4 bits
      );
```

```
end UpDownCounter;
```

Vamos apresentar agora uma **descrição da arquitetura comportamental** da entidade `UpDownCounter` **a nível de funções lógicas**. Neste nível, a entidade é um processo que incrementa ou decrementa o vetor de sinal `counter_reg` de 4 *bits* a cada borda ascendente do sinal `clk`, atualizando o vetor de saída `count` com o valor do sinal interno `counter_reg`.

```
architecture Behavioral of UpDownCounter is
    -- Sinal para armazenar o estado atual do contador (Registrador)
    signal current_count : unsigned(3 downto 0) := (others => '0');
       -- Sinal para calcular o próximo valor do contador (Lógica
Combinacional)
    signal next_count    : unsigned(3 downto 0);
begin

    ---
    ## Processo Síncrono: Registro do Estado
    ---
     -- Este processo é responsável por atualizar o contador na borda de
clock.
    -- Ele infere os flip-flops do contador.
    process(clk, rst)
    begin
        if rst = '1' then
             current_count <= (others => '0'); -- Reset assíncrono para o
registrador
        elsif rising_edge(clk) then
              current_count <= next_count; -- O contador atual recebe o
valor calculado combinacionalmente
        end if;
    end process;

    ---
    ## Processo Combinacional: Lógica do Próximo Estado
    ---
     -- Este processo calcula o próximo valor do contador com base nas
entradas e no estado atual.
    -- Ele é puramente combinacional e infere portas lógicas.
    process(current_count, enable, up_down)
    begin
        -- Valor padrão para evitar latches (mantém o valor atual)
        next_count <= current_count;

        if enable = '1' then -- Habilita a contagem
            if up_down = '1' then
                next_count <= current_count + 1; -- Conta para cima
            else
                next_count <= current_count - 1; -- Conta para baixo
            end if;
        end if;
    end process;

    -- Atribuição da saída: reflete o valor atual do registrador.
    count <= STD_LOGIC_VECTOR(current_count);
```

```
end architecture Behavioral;
```

Observe que em uma atribuição em VHDL, o sinal à esquerda do operador "<=" representa o destino para onde o valor será propagado, enquanto a expressão à direita descreve a operação realizada pelo circuito para gerar esse valor. Além disso, foram utilizados o registrador `current_counter` para armazenar e processar a contagem, em vez de usar o sinal `count`, pois este último será mapeado diretamente nos pinos de saída, sem a função de armazenamento.

Vamos associar à entidade `UpDownCounter` uma segunda arquitetura de **banco de testes** para testar e validar seu funcionamento. A descrição em VHDL de um *testbench* para o contador bidirecional `UpDownCounter` começa com a declaração dos pacotes utilizados e a definição da entidade vazia:

```
-- Code your testbench here
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity UpDownCounter_tb is
end UpDownCounter_tb;
```

Após a declaração da entidade, a arquitetura é definida normalmente, assim como em qualquer módulo VHDL. Os sinais de simulação, `clk`, `rst`, `enable`, `up_down` e `count`, são declarados e inicializados com "0". É importante declarar a instância do(s) módulo(s) que serão testados utilizando a palavra reservada `component`, antes de iniciar a descrição dos testes. Neste caso específico, estamos instanciando o módulo `UpDownCounter`. A definição completa da arquitetura comportamental do contador bidirecional para banco de testes pode ser visualizada a seguir.

```
architecture Behavioral of UpDownCounter_tb is
  signal clk      : STD_LOGIC := '0';
  signal rst      : STD_LOGIC := '0';
  signal enable   : STD_LOGIC := '0';
  signal up_down : STD_LOGIC := '0';
  signal count    : STD_LOGIC_VECTOR(3 downto 0);

  -- Instancia o contador
  component UpDownCounter
      Port (
          clk     : in  STD_LOGIC;
          rst     : in  STD_LOGIC;
          enable   : in  STD_LOGIC;
          up_down : in  STD_LOGIC;
          count    : out STD_LOGIC_VECTOR(3 downto 0)
      );
  end component;

  begin
    -- Instancia o DUT (Device Under Test)
    DUT: UpDownCounter
        Port map (
            clk   => clk,
            rst   => rst,
            enable  => enable,
```

```
            up_down => up_down,
            count   => count
        );

    -- Geração do clock com número limitado de ciclos
    clk_process : process
    begin
        for i in 0 to 100 loop  -- Limite de 100 ciclos de clock
                clk <= '0';
                wait for 10 ns;
                clk <= '1';
                wait for 10 ns;
        end loop;
        wait;  -- Finaliza o processo de clock após o loop
    end process;

    -- Estímulos de teste
    stimulus_process : process
    begin
        -- Reset do contador
        rst <= '1';
        wait for 10 ns;
        rst <= '0';

        -- Testa contagem crescente com enable
        enable <= '1';
        up_down <= '1';
        wait for 100 ns;

        -- Testa contagem decrescente com enable
        up_down <= '0';
        wait for 80 ns;

        -- Desativa enable para testar pausa na contagem
        enable <= '0';
        wait for 40 ns;

        -- Reativa enable
        enable <= '1';
        up_down <= '1';
        wait for 40 ns;

         -- Reduz o tempo de espera adicional para evitar exceder o tempo
    máximo
        wait for 160 ns;

        -- Finaliza a simulação
        wait;
    end process;
  end Behavioral;
```

A primeira seção de descrição de testes, chamada `DUT` (do inglês *Device Under Test*), instancia o componente `UpDownCounter`, mapeando suas portas aos sinais definidos anteriormente. Observe o uso do operador "=>" para realizar o mapeamento de sinais. A segunda seção, `clk_process`, é um processo que gera um sinal de *clock*, alternando entre "0" e "1" a cada 10 ns, durante 100 ciclos. O laço é controlado pelo contador `i`, que garante

que o *clock* será gerado por um número limitado de ciclos. O comando *wait* no final pausa o processo após a geração do *clock*, evitando que ele continue indefinidamente. Por fim, a terceira seção, chamada `stimulus_process`, aplica estímulos ao contador para testar suas funcionalidades. O sinal `rst` é ativado por 10 ns para garantir que o contador comece em zero. Em seguida, o sinal `enable` é ativado e `up_down` é definido como "1" para contagem crescente, com o processo aguardando 100 ns para permitir que a contagem ocorra. O sinal "up_down" é então alterado para "0", mudando a direção da contagem, e o processo aguarda 80 ns. O sinal `enable` é desativado para interromper a contagem por 40 ns, antes de ser ativado novamente, com `up_down` definido como "1" para continuar a contagem crescente, aguardando mais 40 ns. Finalmente, o processo aguarda 160 ns antes de finalizar a simulação com um comando `wait`.

O segundo exemplo é o projeto de semáforo representado em UML. Segue-se uma descrição textual do comportamento do semáforo em VHDL, representado pela entidade `semaforo`. Para melhorar a legibilidade do código, foi definido o tipo enumerado `estado_type` para representar os três estados possíveis: `Red`, `Green` e `Yellow`. O `process(estado_atual, contador)` descreve o comportamento do diagrama de estados UML, com o intervalo de tempo sendo controlado pelo `contador` e pelo valor máximo de contagem, `tempo_max`. Além disso, temos que incluir a descrição do comportamento do circuito de controle do contador, através do `process(clk, reset)`, garantindo que a descrição seja sintetizável em um circuito físico.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity semaforo is
    Port ( clk : in STD_LOGIC;    -- Clock de entrada
    reset : in STD_LOGIC;     -- Reset para reiniciar o semáforo
    cor : out STD_LOGIC_VECTOR(2 downto 0)-- Saída do semáforo (3 bits)
    );
end semaforo;

architecture Behavioral of semaforo is
   -- Definindo os estados da máquina de estados
   type estado_type is (Red, Green, Yellow);
   signal estado_atual, estado_proximo : estado_type;

   -- Definindo o contador de tempo para controle do tempo por estado
   signal contador:integer range 0 to 10 := 0; -- Contador
  constant tempo_max:integer := 5; --Duração de cada estado (ajustável)

begin
    -- Processo de controle de estados
    process(clk, reset)
    begin
        if reset = '1' then
           estado_atual <= Red;  -- Estado inicial é Vermelho
           contador <= 0;
```

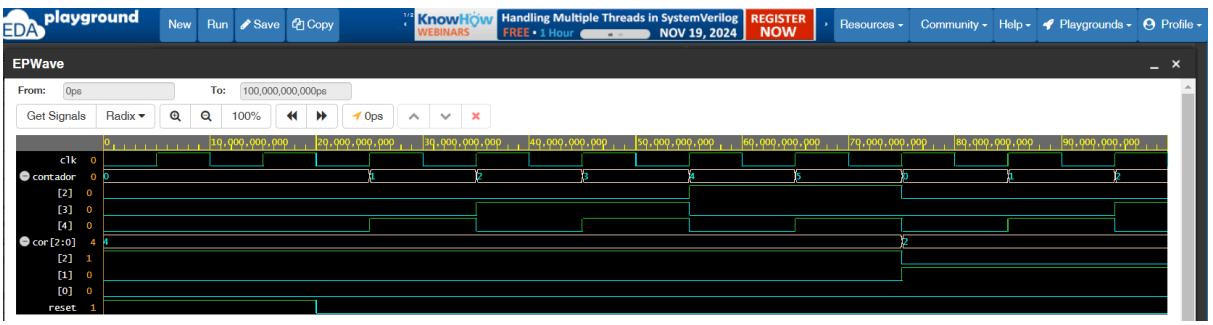
```
        elsif rising_edge(clk) then
           estado_atual <= estado_proximo;  -- Atualiza o estado
           if contador = tempo_max then
              contador <= 0;  -- Ao atingir o tempo máximo: Reinicia
            else
              contador <= contador + 1;  -- Incrementa o contador
            end if;
         end if;
    end process;

    -- Processo de lógica de transição de estados
    process(estado_atual, contador)
    begin
        case estado_atual is
            when Red =>
                cor <= "100";  -- Vermelho
                if contador = tempo_max then
                  estado_proximo <= Green;  -- Transição para Verde
                else
                  estado_proximo <= Red;  -- Continua em Vermelho
                end if;
            when Green =>
                cor <= "010";  -- Verde
                if contador = tempo_max then
                  estado_proximo <= Yellow;  -- Transição para Amarelo
                else
                  estado_proximo <= Green;  -- Continua no estado Verde
                end if;
            when Yellow =>
                cor <= "001";  -- Amarelo
                if contador = tempo_max then
                  estado_proximo <= Red;  -- Transição para Vermelho
                else
                 estado_proximo <= Yellow; -- Continua no estado Amarelo
                end if;
            when others =>
               estado_proximo <= Red; -- Estado de segurança: Vermelho
        end case;
    end process;
end Behavioral;
```

Associamos ainda à entidade `semaforo` uma segunda arquitetura de **banco de testes** para testar e validar seu funcionamento. Segue-se a descrição em VHDL do *testbench*.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity tb_semaforo is
end tb_semaforo;

architecture behavior of tb_semaforo is
    -- Instanciação do componente
    component semaforo
        Port ( clk : in STD_LOGIC;
               reset : in STD_LOGIC;
               cor : out STD_LOGIC_VECTOR(2 downto 0));
```

```
    end component;

    -- Sinais de conexão com a entidade semaforo
    signal clk : STD_LOGIC := '0';
    signal reset : STD_LOGIC := '0';
    signal cor : STD_LOGIC_VECTOR(2 downto 0);

begin
    -- Clock process (periodo de 10 ms)
    clk_process :process
    begin
        clk <= '0';
        wait for 5 ms;
        clk <= '1';
        wait for 5 ms;
    end process;

    -- Instanciando a entidade semaforo
    DUT: semaforo port map (clk => clk,
         reset => reset,
         cor => cor);

    -- Processo de estímulos
    stimulus : process
    begin
        -- Teste de reset
        reset <= '1';
        wait for 20 ms;
        reset <= '0';

        -- Simula o funcionamento do semáforo por 100 ms
        wait for 100 ms;
        -- Finaliza a simulação
        assert false report "Fim da simulação" severity failure;
        wait;
    end process;

end behavior;
```

O resultado da simulação para o intervalo de 100 ms é apresentado a seguir. Após a síntese para um circuito físico, os três sinais do vetor `cor` podem ser utilizados para acionar os circuitos responsáveis pelas três luzes do semáforo.



Correspondências entre esquemáticos e VHDLs

|Vamos mostrar agora como as representações visuais se traduzem em código VHDL, ou vice-versa. A tabela a seguir demonstra essa relação, apresentando uma comparação direta entre os elementos básicos de um esquemático e suas instruções equivalentes em VHDL. Essa correspondência é útil para compreender como um *design* se manifesta tanto graficamente quanto textualmente, e como as ferramentas de síntese interpretam ambos para gerar o *hardware* final.

Elemento esquemático	Descrição	Correspondência em VHDL
Porta AND	Porta lógica com multiplicação booleana	`saida <= entrada1 AND entrada2;`
Porta OR	Porta lógica com adição booleana	`saida <= entrada1 OR entrada2;`
Porta NOT	Inversor lógico	`saida <= NOT entrada;`
Porta NAND	AND seguida de NOT	`saida <= NOT (entrada1 AND entrada2);`
Porta NOR	OR seguida de NOT	`saida <= NOT (entrada1 OR entrada2);`
Porta XOR	Soma exclusiva	`saida <= entrada1 XOR entrada2;`
Porta XNOR	XOR seguida de NOT	`saida <= NOT (entrada1 XOR entrada2);`
Flip-flop tipo D	Armazena o valor na borda de clock	`process(clk) begin if rising_edge(clk) then Q <= D; end if; end process;`
Registrador (vários FF tipo D)	Conjunto de Flip-Flops	`Vetor de sinais: signal reg : std_logic_vector(3 downto 0);` `process(clk) begin if rising_edge(clk)`

		`then reg <= "1001";`
Multiplexador (MUX)	Seleciona uma entre várias entradas	`saida <= A when sel = "00" else B when sel = "01" else ... ;`
Demultiplexador (DEMUX)	Direciona entrada para uma entre várias saídas	Lógica condicional com enable e seletores.
Contador	Incrementa/decrementa valor	`count <= count + 1;` (dentro de `if rising_edge(clk)`)
Comparador	Compara dois valores	`if A = B then ... elsif A > B then ...`
Buffer (ou tri-state)	Isolamento de sinal	`saida <= sinal when enable = '1' else 'Z';`
Bloco de lógica combinacional	Qualquer lógica sem memória	Atribuições diretas com <= (concurrentas ou em process)
Bloco de lógica sequencial	Com memória (depende de *clock*)	Usando `process(clk)` com `if rising_edge(clk)`
Fonte de clock	Sinal periódico de sincronismo	`clk : in std_logic;` (assumido como entrada)
Sinal de reset	Inicializa circuito	`if reset = '1' then ...` (geralmente dentro de `process`)

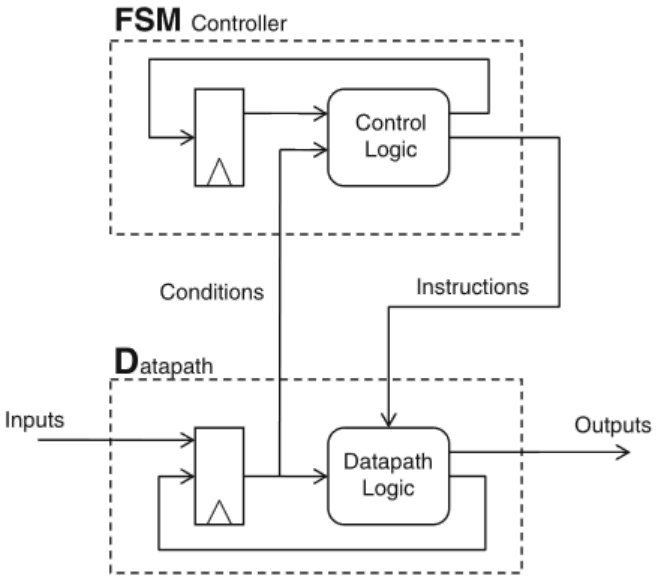
FSMD: UMA METODOLOGIA PARA O PROJETO DE CIRCUITOS DEDICADOS

Dispositivos lógicos programáveis (PLDs), como ASICs, FPGAs e CPLDs, permitem que projetistas desenvolvam circuitos digitais personalizados a partir de descrições em diferentes níveis de abstração, tanto no nível de portas lógicas quanto no nível RTL. No nível RTL, a sincronização do sistema é garantida por um sinal de *clock* comum, que coordena a amostragem e a transferência de dados nos momentos precisos, preservando a integridade do

sistema. A disposição física (*floorplan*) dos componentes dentro dos PLDs é cuidadosamente otimizada para maximizar o desempenho e os caminhos de sinal, enquanto as interconexões entre os blocos de lógica são sintetizadas para atender de maneira eficiente à funcionalidade desejada, oferecendo uma solução altamente personalizada.

Uma metodologia comum para projetar circuitos no nível RTL e sintetizáveis nos PLDs é a metodologia de projeto baseada em RTL, que descreve o sistema por meio de uma sequência de transferências e manipulações de dados entre os registradores. Essa abordagem suporta a execução sequencial de algoritmos e facilita a conversão de um algoritmo em *hardware*, como mostramos no terceiro projeto exemplo `MasterSPI`. Os principais aspectos dessa metodologia incluem:

- o uso de **registradores** para armazenar dados intermediários e representar variáveis do algoritmo ou processo em execução, permitindo a manipulação e transferência de informações ao longo do caminho de dados e assegurando a sincronização das operações sequenciais no sistema,
- a criação de um **caminho de dados** (em inglês, *datapath*) personalizado para realizar as operações necessárias de processamento e manipulação de dados, incluindo o carregamento, armazenamento e transferência entre registradores, e
- a definição de um **caminho de controle** (em inglês, *control path*), frequentemente representado por uma máquina de estados finitos (**FSM**), que especifica a sequência e a lógica de controle para coordenar as operações do sistema, determinando quando e como as operações do caminho de dados devem ser executadas.



A figura mostra como os dois componentes principais de um sistema digital, o caminho de controle (`FSM Controller`) e o caminho de dados (`Datapath`), trabalham em conjunto para coordenar e executar as operações de um sistema digital. O **caminho de controle** é responsável por gerenciar a sequência de operações e garantir a sincronização dos componentes. Ele utiliza **FSMs** para modelar o comportamento sequencial do sistema, descrevendo estados e transições que ocorrem em resposta a eventos, como mudanças nas

entradas ou sinais de controle. A FSM gera sinais de controle que determinam quais operações devem ser realizadas e quando. Esses sinais podem, por exemplo, especificar quando carregar um registrador, quando executar uma operação na ALU, ou qual entrada de um multiplexador deve ser selecionada. O caminho de controle é, portanto, o responsável pela **lógica de controle** (`Control Logic`) e pela orquestração da execução das operações no sistema.

O **caminho de dados**, por sua vez, é sintetizado numa **lógica de caminho de dados** (`Datapath Logic`) responsável por transferir e manipular os dados dentro do sistema. Ele é composto por registradores, ALUs (do inglês *Arithmetic Logic Unit*), multiplexadores (MUXs) e barramentos, que trabalham juntos para processar e transferir os dados de acordo com os sinais `Instructions` recebidos do caminho de controle. Os **registradores,** representados pelos retângulos com um triângulo lateral, armazenam dados intermediários ou variáveis temporárias durante o processamento, sendo atualizados conforme as instruções do caminho de controle. A ALU é responsável por executar operações aritméticas e lógicas sobre os dados, como somar, subtrair, ou realizar operações AND/OR. O caminho de controle determina quando a ALU deve executar essas operações, enviando os sinais de controle adequados. Os multiplexadores (MUXs) são usados para selecionar quais dados serão passados entre os diferentes componentes, como registradores e ALUs, com base nos sinais de controle. Já os barramentos são responsáveis por transportar os dados entre os registradores, a ALU e outros elementos do sistema.

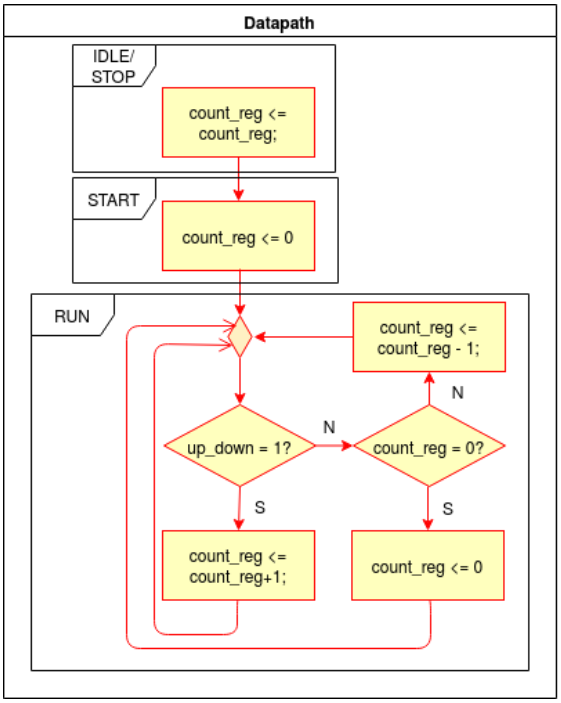
Ambos os caminhos são sincronizados pelo sinal de *clock*, que define os momentos exatos em que as transições devem ocorrer, garantindo que as operações sejam executadas no momento adequado e de forma coordenada. A interação entre o caminho de controle e o caminho de dados é fundamental para o funcionamento de um sistema digital, e essa interação é **bidirecional**. Além dos sinais de controle gerados pelo caminho de controle (`Instructions`) que comandam as operações no caminho de dados, os dados processados podem gerar resultados que influenciam o comportamento do caminho de controle. Esses resultados (`Conditions`) podem ser *flags*, códigos de erro ou resultados intermediários de operações realizadas, e são enviados de volta ao caminho de controle para serem levados em consideração nas decisões sobre o fluxo de execução. Essa troca de informações permite que o sistema se ajuste dinamicamente durante o processamento, reagindo a condições específicas que surgem ao longo da execução, e garantindo maior flexibilidade e adaptabilidade ao comportamento do circuito.

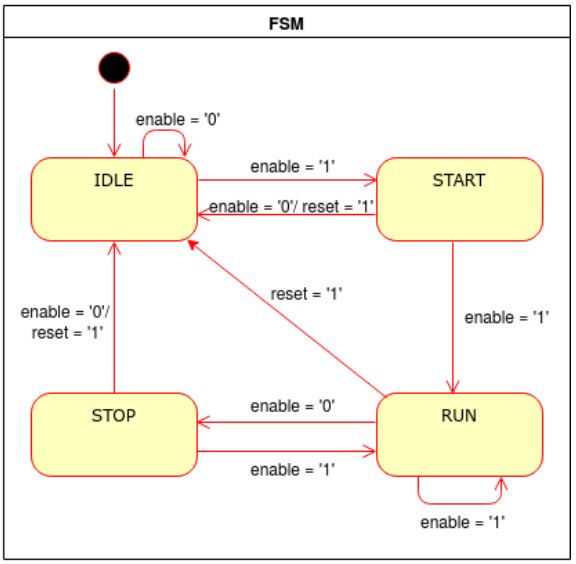
Essa abordagem de combinar a máquina de estados finitos (FSM) com o caminho de dados de forma coordenada é conhecida como **FSM com caminho de dados** (em inglês, *Finite-State Machine with Datapath* – **FSMD**). Em vez de ter uma FSM isolada controlando o sistema e um caminho de dados independente, a FSMD integra ambos, de forma que a FSM controla o fluxo de dados no caminho de dados enquanto gerencia as transições de estados. Ao mesmo tempo, o caminho de dados executa as operações reais sobre os dados, com base nas instruções ou sinais de controle enviados pela FSM. Essa arquitetura unifica a lógica de controle e a estrutura de manipulação de dados em um modelo coeso e eficiente, amplamente

utilizado no desenvolvimento de sistemas digitais dedicados. Ao integrar controle e processamento de dados de forma coordenada, a FSMD permite a implementação de algoritmos complexos diretamente em *hardware*, proporcionando maior desempenho e previsibilidade temporal em aplicações embarcadas.

No contexto de circuitos digitais, um exemplo clássico da aplicação de FSMD é o projeto de contadores. O exemplo a seguir ilustra como a combinação de uma FSMD pode ser aplicada ao projeto de um contador bidirecional de 4 *bits* com 4 pinos de entrada (*enable*, *reset*, *clock* e *up_down*) e 4 pinos de saída (*count*), cuja descrição em VHDL, sem o uso explícito de estados, foi apresentada anteriormente. Neste projeto, a FSM controla os quatro estados do contador: `IDLE` (ocioso), `START` (início), `RUN` (execução) e `STOP` (parada). O caminho de dados, por sua vez, é responsável por manipular os valores armazenados no contador, realizando as operações de incremento, decremento ou mantendo o valor atual, conforme necessário.

O diagrama de estados da FSM, juntamente com o diagrama de atividades do caminho de dados associados aos quatro estados do circuito, é apresentado na figura a seguir. Vale destacar que cada estado é caracterizado por uma atualização específica do conteúdo do registrador `count_reg`, cujo valor é refletido nos pinos de saída `count`. Além disso, devido à separação clara entre a FSM e o caminhho de dados, torna-se possível modificar a lógica de processamento de dados (como as operações de contagem) com poucas ou nenhuma alteração na estrutura de controle de estados, o que confere flexibilidade e modularidade ao projeto.





apresenta-se uma descrição da representação gráfica em VHDL, sintetizável em dispositivos lógicos programáveis. A arquitetura `FSM_with_Datapath` é composta por dois processos,

um processo `fsm`, responsável pelo controle dos estados e transições, e um processo `datapath`, que executa as operações de contagem bidirecional. O valor do contador é armazenado no registrador `count_reg` e é passado para a saída `count`.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity contador is
    Port ( clk     : in STD_LOGIC;
           reset   : in STD_LOGIC;
           enable  : in STD_LOGIC;
           up_down : in STD_LOGIC;
           count   : out STD_LOGIC_VECTOR(4 downto 0));
end contador;

architecture FSM_with_Datapath of contador is
  type state_type is (IDLE, START, RUN, STOP);
    signal current_state : state_type := IDLE;
    signal count_reg :STD_LOGIC_VECTOR(4 downto 0);

begin
    -- Processo de controle da FSM
    fsm: process(clk, reset)
    begin
        if reset = '1' then
            current_state <= IDLE;
        elsif rising_edge(clk) then
            case current_state is
              when IDLE =>
                    if enable = '1' then
                    current_state <= START;
                   end if;
              when START =>
                    if enable = '1' then
                    current_state <= RUN;
                   else
                    current_state <= IDLE;
                   end if;
                when RUN =>
                    if enable = '0' then
                    current_state <= STOP;
                   end if;
                when STOP =>
                    if enable = '1' then
                    current_state <= RUN;
                   else
                    current_state <= IDLE;
                   end if;
            end case;
        end if;
    end process;

    -- Lógica do datapath
    datapath: process(clk, current_state)
```

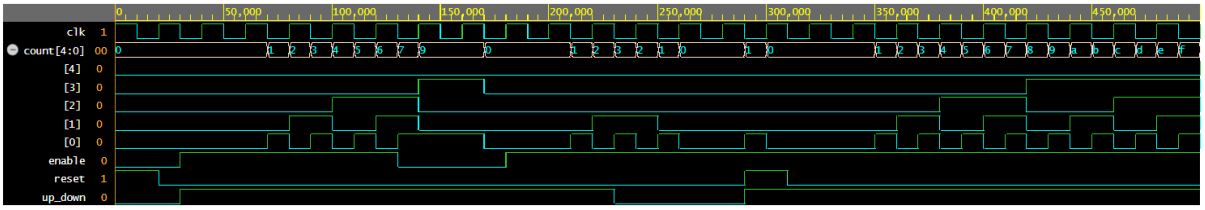
```
    begin
        case current_state is
        when IDLE =>
         count_reg <= (others => '0'); -- Reseta
        when RUN =>
          if up_down = '1' then
             count_reg <= count_reg + 1;  -- crescente
          else
            if count_reg = 0 then
             count_reg <= count_reg;
            else
             count_reg <= count_reg - 1;  -- decrescente
            end if;
          end if;
        when others =>
         count_reg <= count_reg;
        end case;

        count <= count_reg; -- Saída do contador
    end process;

end FSM_with_Datapath;
```

Para verificação e validação do comportamento do circuito descrito, foram executadas simulações com o seguinte banco de testes.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.NUMERIC_STD.ALL;

-- Entidade Testbench
entity tb_contador is
end tb_contador;

-- Arquitetura do Testbench
architecture behavior of tb_contador is
    -- Sinais para conectar com a unidade DUT (Device Under Test)
    signal clk     : STD_LOGIC := '0';
    signal reset   : STD_LOGIC := '0';
    signal enable  : STD_LOGIC := '0';
    signal up_down : STD_LOGIC := '0';
    signal count   : STD_LOGIC_VECTOR(4 downto 0);

    -- Componente DUT (Contador_4bits)
    component contador is
        Port ( clk     : in STD_LOGIC;
               reset   : in STD_LOGIC;
               enable  : in STD_LOGIC;
               up_down : in  STD_LOGIC;
               count   : out STD_LOGIC_VECTOR(4 downto 0));
    end component;

begin
    -- Instanciação do DUT (Contador_4bits)
    DUT: contador
        port map (
            clk => clk,
```

```
            reset => reset,
            enable => enable,
            up_down => up_down,
            count => count
        );

    -- Processo para gerar o clock
    clock_process : process
    begin
        -- Gera um clock com período de 10ns (frequência de 50MHz)
        clk <= not clk;
        wait for 10 ns;
    end process;

    -- Processo para aplicar estímulos de teste
    stimulus_process : process
    begin
        -- Teste 1: Reseta o contador
        reset <= '1';
        wait for 20 ns;
        reset <= '0';
        wait for 10 ns;

        -- Teste 2: Habilita contagem
        enable <= '1';
        up_down <= '1';
        wait for 100 ns; -- Deixe o contador rodar por 100ns
        enable <= '0';
        wait for 20 ns;

        -- Teste 3: Verifica o contador desabilitado
        enable <= '0';
        wait for 30 ns;

        -- Teste 4: Habilita contagem novamente
        enable <= '1';
        wait for 50 ns;

        -- Teste 5: Contagem decrescente
        up_down <= '0';
        wait for 60 ns;

        -- Teste 5: Reseta o contador no meio da contagem
        reset <= '1';
        up_down <= '1';
        wait for 20 ns;
        reset <= '0';
        wait for 10 ns;

        -- Finaliza a simulação
        wait;
    end process;
end behavior;
```

As formas de onda geradas pelas simulações confirmam que o comportamento do circuito está em conformidade com a especificação do projeto.



No projeto-exemplo `MasterSPI`, vimos que a FSM é definida na entidade `StateMachine`, com os seguintes estados: `IDLE`, `PREPARE`, `SCK1`, `SCK0` e `SAVE`. Cada estado gera um vetor de 8 *bits* chamado `out_reg`, que controla as operações do *datapath* via sinais `LDTx`, `SHTx`, `RSRx`, `SHRx`, `LDRx`, `RSCNT`, `TCPLT` e `SCK`:

Estado FSM	Sinais ativos em Ctrl_Internal	Efeito no caminho de dados
IDLE	00000010	Ativa apenas TCPLT = 1 → indica fim da transmissão. Nenhuma operação no *datapath*.
PREPARE	10100100	LDTx=1 → carrega registrador de shift Tx RSRx=1 → reseta Rx shift RSCNT=1 → reseta contador de *bits*.
SCK1	00010001	SHRx=1 e SCK= 1 → desloca dado no Rx e emite *clock*.
SCK0	01000000	SHTx=1 → desloca dado no Tx.
SAVE	00001000	LDRx=1 → carrega dados paralelos recebidos no registrador final Rx.

Projeto de um processador dedicado

A metodologia de projeto FSMD permite uma modelagem clara e modular, combinando máquinas de estados finitos (FSM) com um caminho de dados (em inglês, *datapath*) que processa as informações de forma eficiente. O procedimento para desenvolver um processador dedicado aplicando esta metodologia, com o objetivo de implementação física por meio de HDL, envolve várias etapas essenciais para garantir que o projeto seja funcional e eficiente. Abaixo estão as etapas principais:

1. **Identificação dos Requisitos**: Antes de iniciar a modelagem do sistema, identifique os requisitos funcionais e de desempenho. Isso envolve entender detalhadamente as operações que o *hardware* deverá realizar e as especificações de temporização (*timing*) exigidas. Nessa fase, define-se o que o sistema deve realizar, ou seja o **procedimento** ou **algoritmo** a ser implementado, quais dados serão processados e como o desempenho será avaliado.
2. **Definição da Arquitetura**: Com os requisitos bem definidos, a próxima etapa é definir a arquitetura do sistema. Isso inclui a seleção de componentes principais, como registradores, multiplexadores, ALUs, unidades de controle e barramentos, além de estabelecer como eles interagirão. A definição da arquitetura envolve estruturar o

fluxo de dados e as interações entre os componentes do projeto, além de assegurar que o projeto atenda aos requisitos de performance e funcionalidade.

3. **Definição do Caminho de Dados**: A partir do procedimento elaborado, constroem-se os **caminhos de dados** do sistema. O primeiro passo é listar todas as operações de transferência entre registradores. Em seguida, agrupe essas operações conforme os registradores de destino envolvidos. Para cada grupo de operações, inicie pela construção do registrador de destino e pelo projeto dos circuitos combinacionais necessários para realizar as operações associadas a ele. Se o registrador de destino estiver envolvido em múltiplas operações, adicione circuitos de multiplexação e roteamento para garantir que os sinais corretos sejam encaminhados ao registrador adequado. Além disso, considere os circuitos responsáveis por gerar os sinais de estado (em inglês, *status signals*) necessários para o controle e monitoramento das operações.
4. **Modelagem dos Estados**: A modelagem dos estados do sistema é uma etapa fundamental, especialmente em sistemas sequenciais. Identificar os estados e as relações entre eles ajuda a organizar a execução das operações de forma estruturada. A partir dos caminhos de dados, pode-se utilizar os registradores de destino como uma base para dividir os **estados** do sistema. Para cada estado, defina qual tarefa específica, ou ações específicas, ele deve realizar, como carregar um registrador, realizar uma operação aritmética ou esperar uma entrada.
5. **Estabelecimento das Transições entre Estados**: Após identificar os estados, é necessário definir como eles se relacionam entre si. Para cada estado, identifique as condições ou eventos necessários para que o sistema **transite** para outros estados. Essas condições podem incluir entradas externas, como sinais de controle ou dados de entrada do sistema; sinais de estado internos gerados pelo caminho de dados, como *flags*; além de contadores ou temporizadores, que ajudam a sincronizar operações ou a garantir que certas ações ocorram após um período específico. Uma prática comum é representar essa estrutura usando tabelas de transição ou diagramas de estados, que facilitam a visualização das transições entre estados com base em entradas, sinais de controle e condições internas.
6. **Codificação em HDL**: Com a arquitetura e as relações entre os estados claramente definidas, o próximo passo é codificar o projeto em uma linguagem de descrição de *hardware*, como VHDL ou Verilog. Nesta fase, é importante traduzir a arquitetura e o comportamento do sistema para a HDL de forma que a implementação corresponda ao modelo RTL. Deve-se considerar as operações de transferência de dados entre registradores, controle das operações aritméticas e lógicas, e a interação entre os diferentes componentes.
7. **Simulação e Verificação**: Após a codificação, é essencial simular o projeto para verificar se ele atende aos requisitos funcionais e de temporização. A simulação deve incluir testes de funcionalidade, verificando se o comportamento do circuito está correto. Além disso, a verificação de temporização (*timing*) deve ser realizada para garantir que todas as operações aconteçam no tempo correto e que não haja violações de *timing*, o que pode comprometer a operação do sistema. Análise de cobertura de

teste também pode ser utilizada para garantir que todos os cenários possíveis foram testados.

8. **Síntese e Implementação**: Após a simulação bem-sucedida, o projeto passa pela fase de síntese, onde a descrição HDL é convertida em uma descrição de nível de porta e blocos funcionais. Isso cria a representação física do circuito, que pode ser implementada em dispositivos como FPGAs ou ASICs. A síntese é responsável por otimizar o *design* de acordo com os requisitos de área e performance.
9. **Testes Físicos e Validação**: Após a implementação, o projeto sintetizado deve ser testado fisicamente no dispositivo de destino (FPGA ou ASIC) para garantir que ele funcione conforme o esperado nas condições reais de operação. A validação em *hardware* pode incluir a execução de testes adicionais de funcionalidade e análise de desempenho, assegurando que o sistema atenda às especificações de desempenho sob diferentes condições de operação.

Para ilustrar o procedimento, o projeto-exemplo `MasterSPI` apresenta o desenvolvimento de um módulo de comunicação serial síncrona SPI. É demonstrado como a metodologia FSMD pode ser aplicada para criar soluções de *hardware* através das simulações.