

LaPSE

Laboratório de Prototipação Digital
e Sistemas Embarcados

Universidade do Vale do Rio dos Sinos - UNISINOS

Prototipação em *PLDs*

Introdução ao *VHDL*

Autor: Prof. Rodrigo Marques de Figueiredo

Agenda

- Introdução;
 - Histórico;
 - Características do *VHDL*;
 - Aplicações do *VHDL*;
 - Novas Tecnologias.
- Estrutura da Linguagem;
- Implementações de Lógica Assíncronas;
- ModelSim (Simulador);
- Implementação de Lógica Síncrona;
- Integração Entre Módulos Descritos em *VHDL*;
- *Testbenches* Descritos em *VHDL*.

VHDL

Introdução

Histórico

- **1968** – primeiras *HDLs* (incompatíveis);
- **1973** – primeiro esforços de padronização (*CONLAN*) início do primeiro projeto patrocinado pelo *DoD* (*Department of Defence*);
- **1983** – final do *CONLAN* (*ADA*) e *DoD* inicia o projeto *VHSIC* com um consórcio de empresas privadas (*IBM*, *Intermetrics* e *Texas Instruments*);
- **1986** – primeiros compilador e simulador e criação de um grupo de padronização do *IEEE* para *VHDL*;

Histórico

- **1988** – primeiros softwares são comercializados;
- **1991** – novo processo de padronização;
- **1993** – um novo padrão é publicado, chamado *VHDL'93*, padronizado *IEEE Std 1164-1993*;
- **1997** – foi publicado o manual de referência da linguagem *VHDL*;
- **2002** – Lançado o *VHDL-AMS* (*VHDL para Analog and Mixed Signals*);
- **2003** – publicado o novo padrão chamado *VHDL'03*, padronizado *IEEE Std 1164-1993*.

Características do VHDL

- Características:
 - Modular;
 - Paralelismo entre instruções (comandos);
 - Fácil de documentar.
- Desvantagens:
 - VHDL não gera um hardware otimizado.

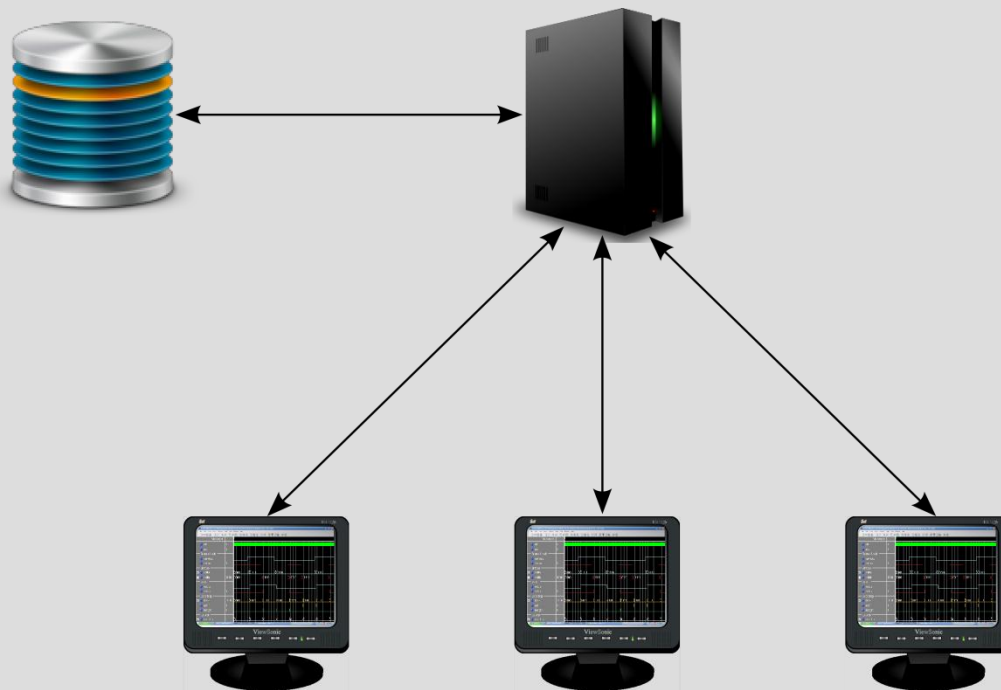
Características do VHDL

■ Vantagens:

- Intercâmbio de projetos entre grupos de pesquisa sem a necessidade de alteração;
- A linguagem independe da tecnologia atual, ou seja, você pode desenvolver um sistema hoje e implementá-lo depois;
- Os projetos são **fáceis** de serem modificados;
- O custo de produção de um circuito dedicado é elevado, enquanto que usando VHDL e Dispositivos Programáveis, isto passa a ser muito menor;
- Reduz consideravelmente o tempo de projeto e implementação.

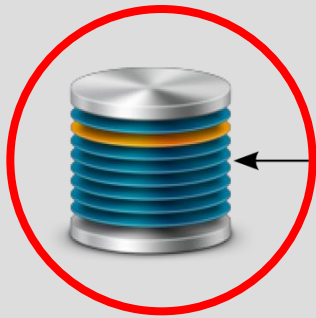
Características do VHDL

- Vantagens:

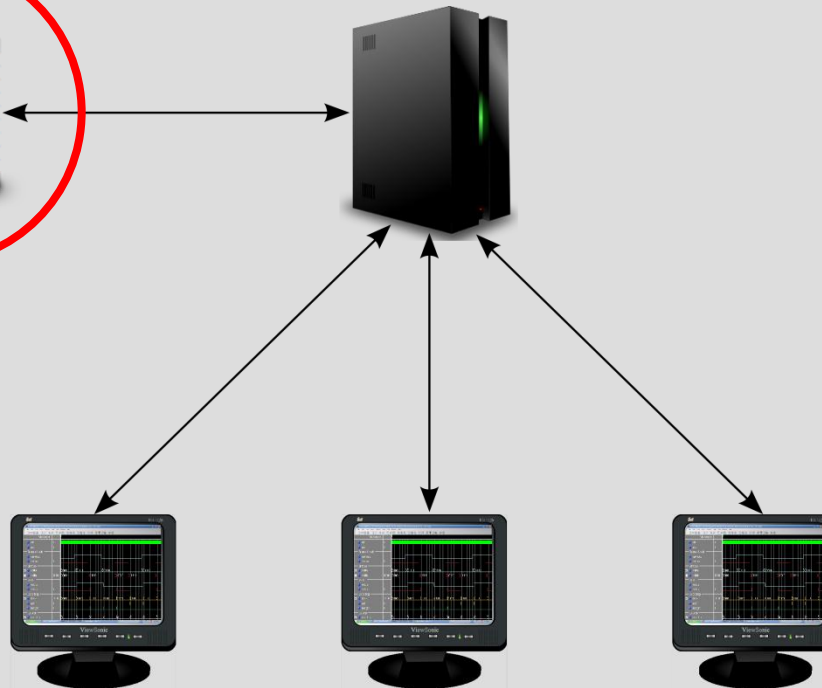


Características do VHDL

■ Vantagens:

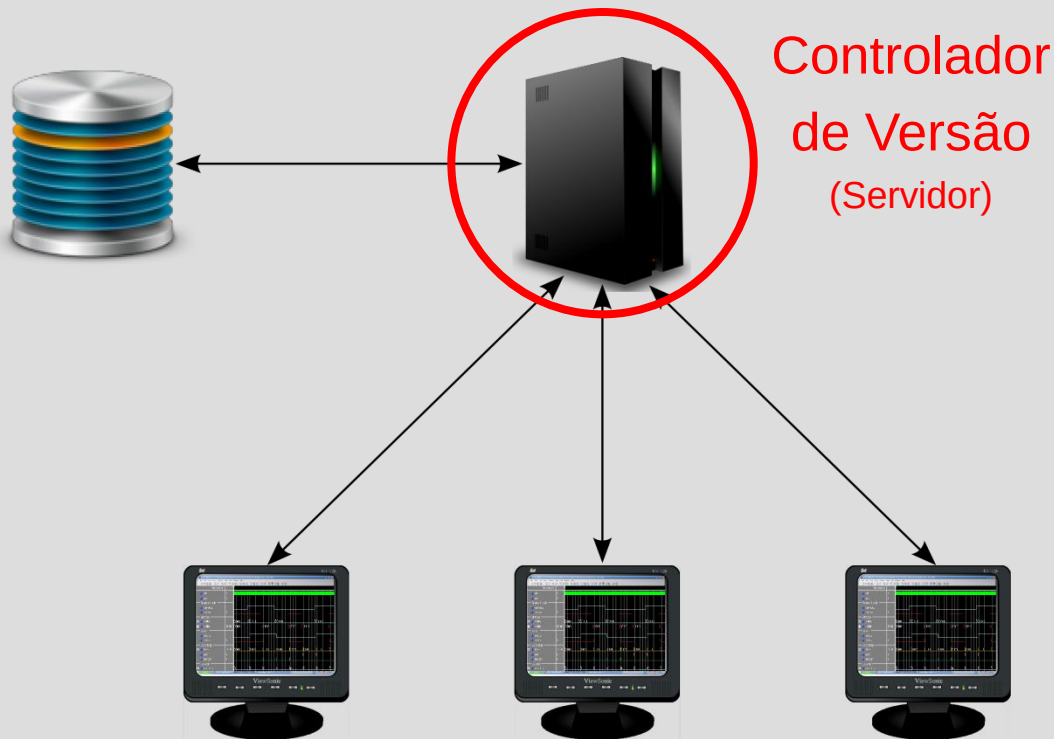


Base de Cores
(componentes em VHDL)



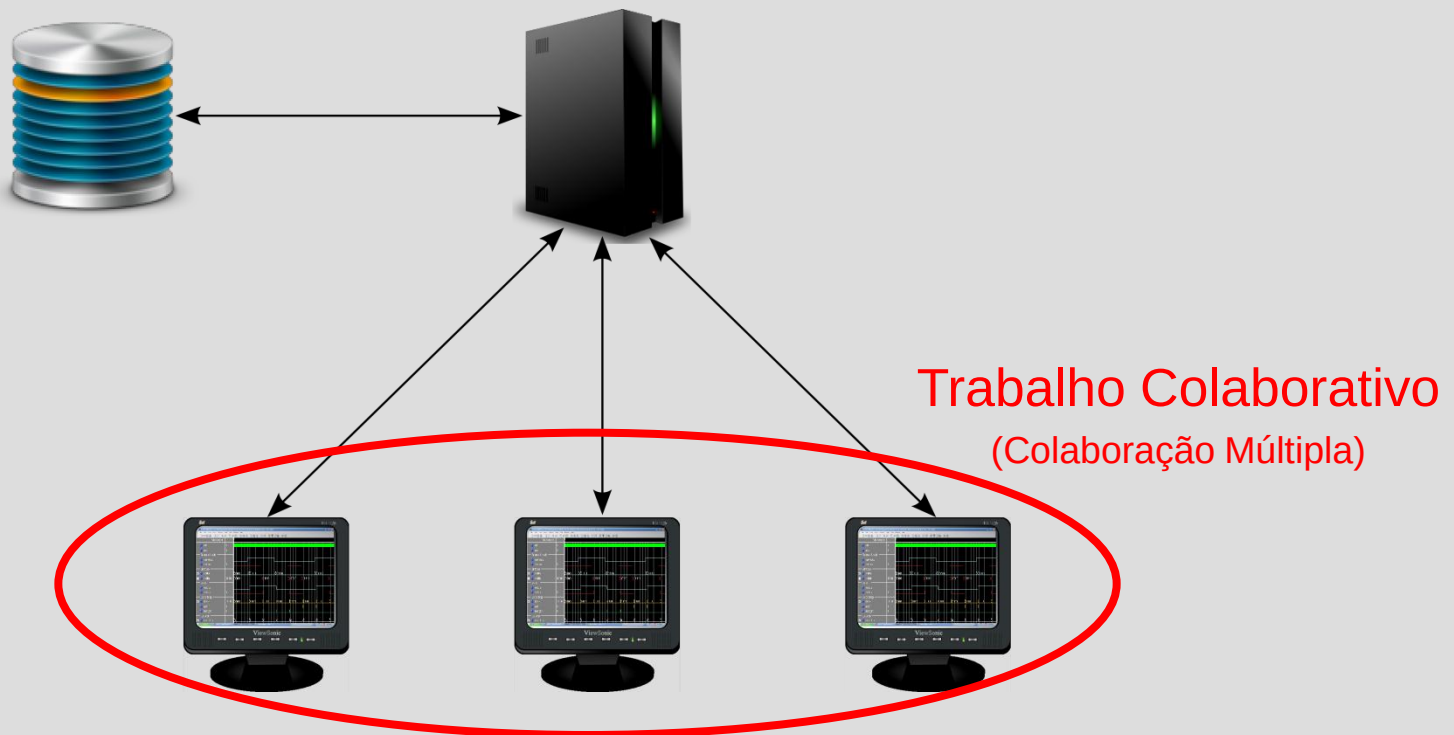
Características do VHDL

■ Vantagens:



Características do VHDL

- Vantagens:



Aplicações do VHDL

- Controle de Processos;
- Instrumentação;
- *Drivers* de Barramentos;
- Conversores de Interface;
- Novas aplicações;
- Processamento Digital de Sinais:
 - Processamento de Áudio;
 - Processamento de Vídeo;
 - Multiprocessamento de sinais de controle (Controle Multivariável).

Novas Tecnologias

- Cores de Microcontroladores *Softcores*;
- *DSP* com auxílio de pacote do *MatLab*;
- *PLDs* transceivers;
- Soluções para projetos com circuitos analógicos e mistos;
- *CAD* para projetos de *hardware* dedicados (*Block RAMs*, *Distributed RAMs*, *Digital Clock Manager*, ...);
- Ferramentas para verificação de ocupação (efetiva) do *PLD*;
- *VHDL-AMS* (*VHDL* para *Analog and Mixed Signals*).

Tipos de Descrições

■ Descrição Comportamental:

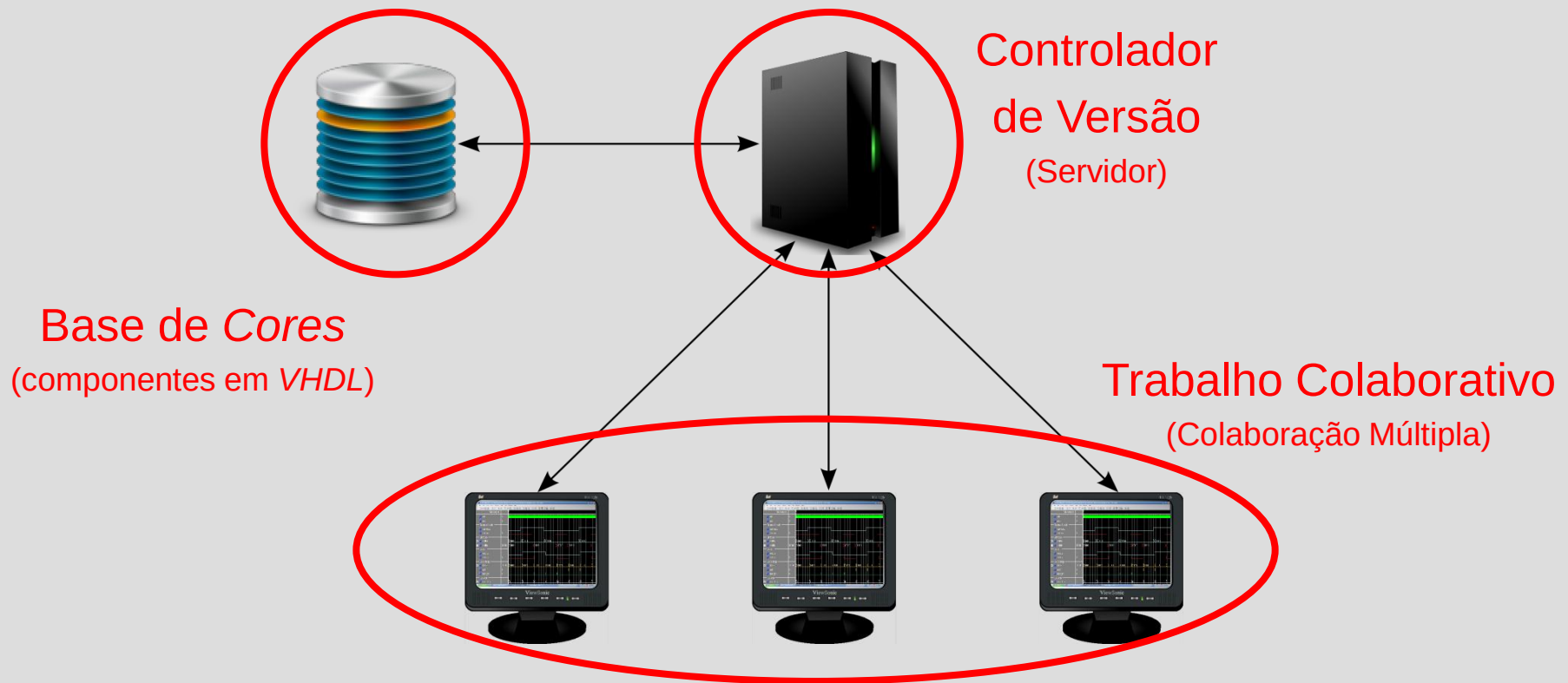
- Mais utilizada por ter curva de aprendizado mais 'rápida';
- Seu principal problema é ser dependente da descrição realizada, por sua vez dependente da *expertise* do projetista;
- Sua principal virtude é a velocidade de implementação de uma funcionalidade.

■ Descrição Estrutural:

- Menos utilizada por ter curva de aprendizado mais 'lenta';
- Sua principal virtude é o controle do *hardware* gerado pela descrição;
- Seu principal problema é seu tempo de implementação;
- Porém pode ser contornado por um bom gerenciamento de projeto.

Características do VHDL

■ Vantagens:



Estrutura da Linguagem

Estrutura da Linguagem

Um “programa” em *VHDL* possui 3 estruturas básicas e 1 sub-estrutura:

- **Package;**
- **Entity;**
- **Architecture & Process.**

<pre>library IEEE; use ieee.std_logic_1164.all; use ieee.std_logic_unsigned.all;</pre>	Package (Bibliotecas)
<pre>entity Clxx is port (-- descrição dos pinos de entrada e saída); end Clxx ;</pre>	Entity (Pinagem E/S)
<pre>architecture comportamento of Clxx is begin process(-- pinos de entrada e signal) begin -- descrição do circuito integrado end process; end comportamento ;</pre>	Architecture (Descrição do comportamento do componente)



Estrutura da Linguagem

■ **Package:**

```
-- declaração das libraries  
package nome_do_package is  
  -- declaração de tipos (opcional)  
  -- declaração de subtipos (opcional)  
  -- declaração de constantes (opcional)  
  -- declaração de sinais (opcional)  
  -- declaração de componentes (opcional)  
  -- declaração de funções (opcional)  
end nome_do_package;
```

Estrutura da Linguagem

- **Entity:**

```
entity nome_topo_de_hierarquia is
  port (
    reset          : in  std_logic;
    clock          : in  std_logic;
    entrada        : in  std_logic_vector(7 downto 0);
    saida          : out std_logic_vector(7 downto 0)
  );
end nome_topo_de_hierarquia;
```

As portas podem ser declaradas com *in*, *out*, *inout* e *buffer*.

Estrutura da Linguagem

■ *Entity*:

```
entity nome_topo_de_hierarquia is  
  generic(  
    TAMANHO : integer := 8  
  );  
  port (  
    reset          : in  std_logic;  
    clock           : in  std_logic;  
    entrada         : in  std_logic_vector(TAMANHO-1 downto 0);  
    saida           : out std_logic_vector(TAMANHO-1 downto 0)  
  );  
end nome_topo_de_hierarquia;
```

Os *generics* são utilizados para parametrizar os códigos.

Estrutura da Linguagem

■ **Architecture:**

```
architecture Nome_da_Arquitetura of Nome_da_Entidade is  
    -- Declaração de sinais e tipos próprios (ou variáveis)  
begin  
    -- Descrição da funcionalidade (Coportamento) da aplicação  
end Nome_da_Arquitetura;
```

- É dentro (após a palavra reservada *begin*) da *architecture* que é descrita a lógica do programa propriamente dita;
- Podem ser declarados *process*, que são executados em paralelo dentro da *architecture*.

Estrutura da Linguagem

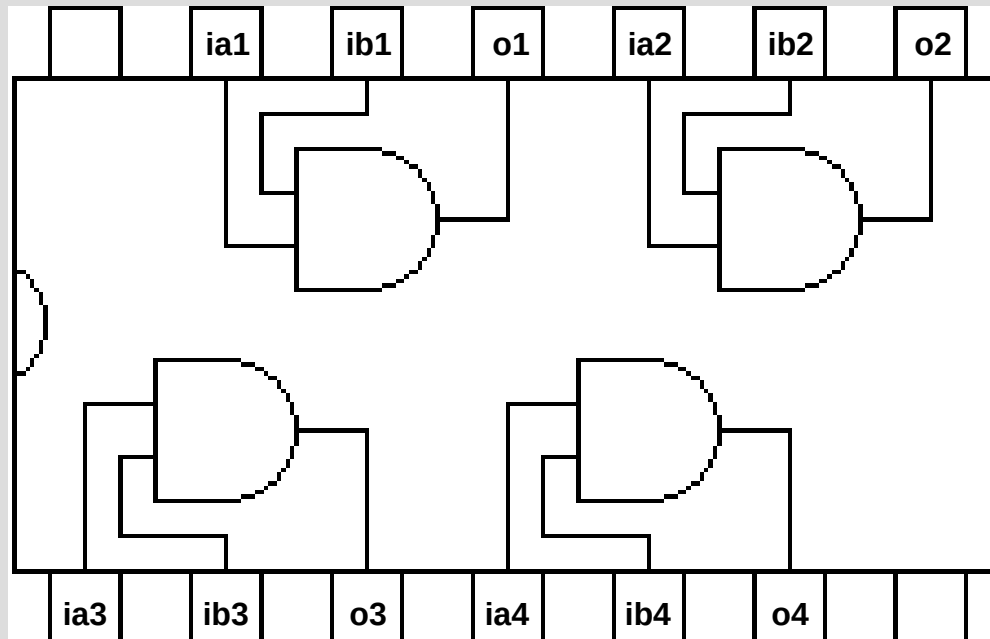
■ **Process:**

```
Nome_do_processo:  
process(-- Lista de sensibilidade)  
    -- Declaração de variáveis  
    begin  
        -- Descrição da lógica utilizada  
    end process Nome_do_processo;
```

- Toda lógica descrita fora de um *process* é assíncrona;
- Lógicas síncronas devem ser descritas obrigatoriamente dentro de um *process*;
- Todos os sinais analisados pelo processo devem constar na lista de sensibilidade, sob pena de não ser inferido na implementação.

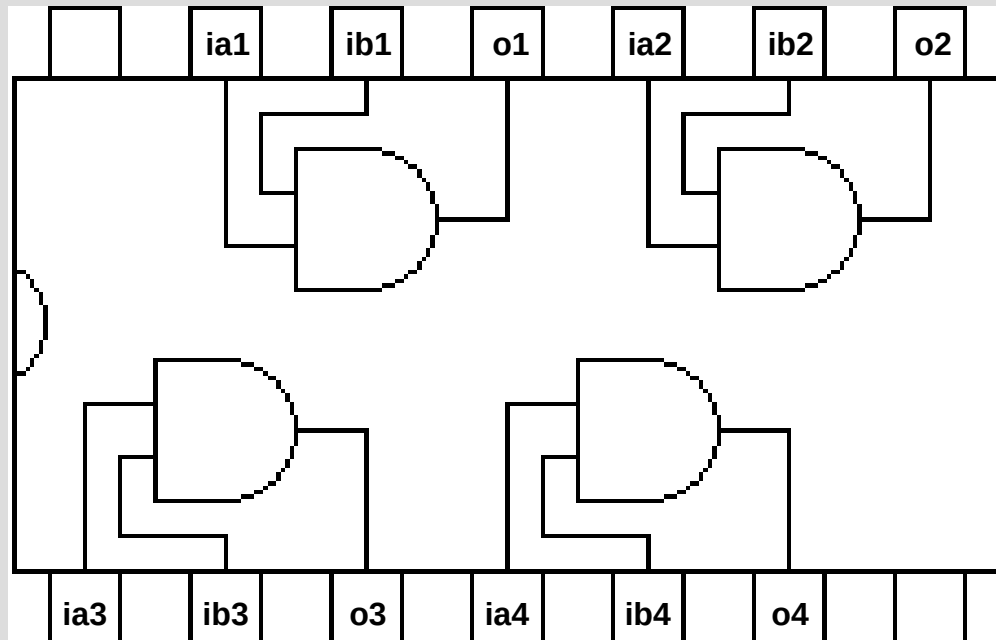
Estrutura da Linguagem

- Implementação:



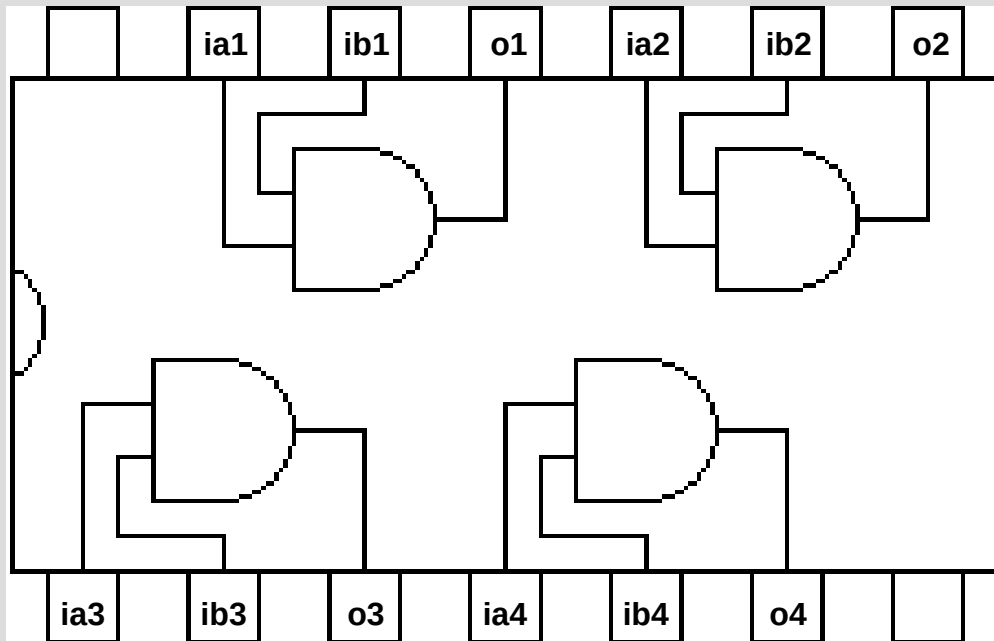
Estrutura da Linguagem

■ Implementação:



Estrutura da Linguagem

■ Implementação:



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

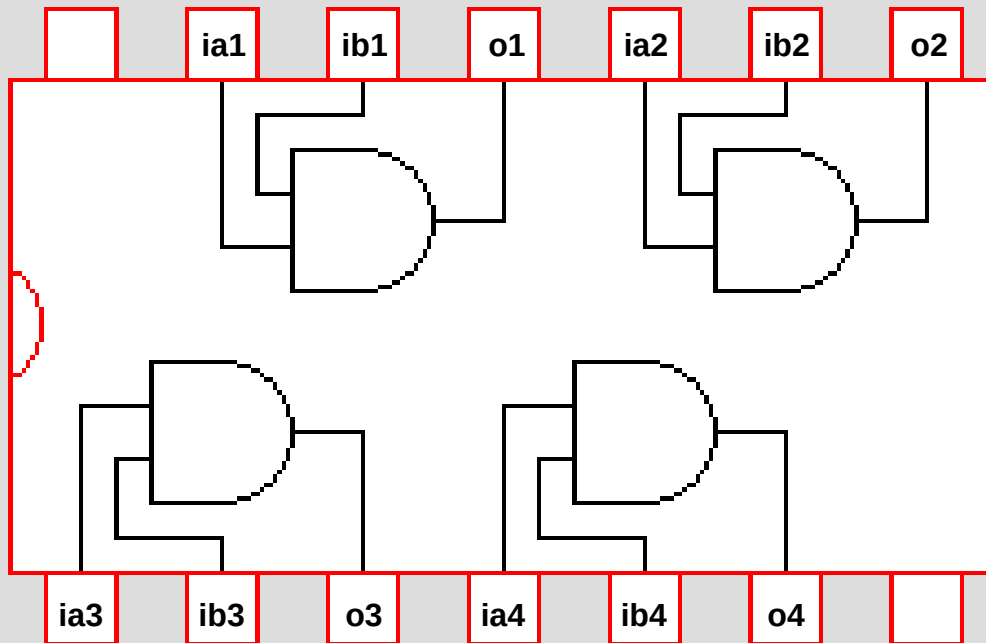
-- Entidade
-----
entity CI7408 is
  port (
    ia1 : in    std_logic;
    ib1 : in    std_logic;
    o1  : out   std_logic;
    ia2 : in    std_logic;
    ib2 : in    std_logic;
    o2  : out   std_logic;
    ia3 : in    std_logic;
    ib3 : in    std_logic;
    o3  : out   std_logic;
    ia4 : in    std_logic;
    ib4 : in    std_logic;
    o4  : out   std_logic
  );
end CI7408;

-- Arquitetura
-----
architecture CI7408 of CI7408 is
begin
  o1 <= ia1 and ib1;
  o2 <= ia2 and ib2;
  o3 <= ia3 and ib3;
  o4 <= ia4 and ib4;

end CI7408;
```

Estrutura da Linguagem

■ Implementação:



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

```
-- Entidade
-----
```

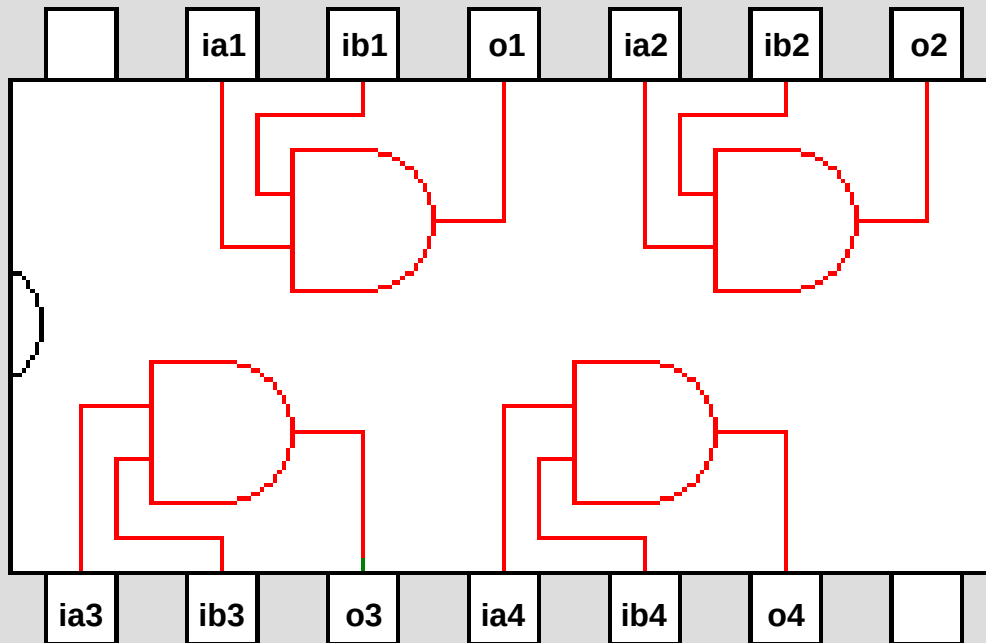
```
entity CI7408 is
  port (
    ia1 : in    std_logic;
    ib1 : in    std_logic;
    o1  : out   std_logic;
    ia2 : in    std_logic;
    ib2 : in    std_logic;
    o2  : out   std_logic;
    ia3 : in    std_logic;
    ib3 : in    std_logic;
    o3  : out   std_logic;
    ia4 : in    std_logic;
    ib4 : in    std_logic;
    o4  : out   std_logic
  );
end CI7408;
```

```
-- Arquitetura
-----
```

```
architecture CI7408 of CI7408 is
begin
  o1 <= ia1 and ib1;
  o2 <= ia2 and ib2;
  o3 <= ia3 and ib3;
  o4 <= ia4 and ib4;
end CI7408;
```

Estrutura da Linguagem

■ Implementação:



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

-- Entidade

entity CI7408 is
    port (
        ia1 : in    std_logic;
        ib1 : in    std_logic;
        o1  : out   std_logic;
        ia2 : in    std_logic;
        ib2 : in    std_logic;
        o2  : out   std_logic;
        ia3 : in    std_logic;
        ib3 : in    std_logic;
        o3  : out   std_logic;
        ia4 : in    std_logic;
        ib4 : in    std_logic;
        o4  : out   std_logic
    );
end CI7408;

-- Arquitetura

architecture CI7408 of CI7408 is
begin
    o1 <= ia1 and ib1;
    o2 <= ia2 and ib2;
    o3 <= ia3 and ib3;
    o4 <= ia4 and ib4;

end CI7408;
```

Declarações

■ ***Signals, Constants & Variables:***

• ***Signals:***

- Os *signals* são declarados antes do início (*begin*) da *architecture*;
- São “visíveis” por todos os processos da *architecture*;
- Podem ser vistos como os “fios” que unem os diversos processos;
- Sua prototipação é mais estável.

```
signal nome_do_sinal : std_logic;
```

Declarações

■ *Signals, Constants & Variables:*

• *Constants:*

- As *constants* são declaradas a exemplo dos **signals** dentro da **architecture**, antes de seu início (**begin**);
- São visíveis por todos processos a que pertencem;
- Geralmente são utilizadas para facilitar a documentação do código.

```
constant nome_da_constante : std_logic := '1';
```

Assignments

- ***Signals & Variables:***

- ***Signals:***

`nome_do_sinal <= '1';`

- ***Variables:***

`nome_da_variável := '0';`

O sinal de assignment sempre será do tipo do identificador da esquerda.

Operadores

■ Operadores Lógicos:

- *Em VHDL estão disponíveis todos os operadores lógicos existentes em lógica discreta: NOT, AND, NAND, OR, NOR, XOR e XNOR.*

■ Operadores de Comparação:

- *Estes operadores são fundamentais para as implementações de lógicas computacionais, sendo eles:*

=	igual
/=	diferente
>	maior que
>=	maior igual a
<	menor que
<=	menor igual a

Operadores

■ Operadores Aritméticos:

- São operadores muito utilizados para auxiliar na parametrização de códigos ou nos *testbenches*;
- Geralmente utilizados em tempo de síntese e não de execução;
- Quando utilizados em tempo de execução estes operadores são performados por **macros** que geram um *hardware* muito longe do ideal.

+	Adição
-	Subtração
*	Multiplicação
/	Divisão
**	potenciação

&	concatenação
mod	módulo
rem	resto da divisão
abs	valor absoluto

Números

■ Constantes:

- Existe mais de uma maneira expressar números em VHDL, seguem algumas:

Base#número# ➔ **16#ACDC#**

letra“string” ➔ **x“ACDC”** ou **b“10101010”** ou **o“755”**

decimal ➔ **255** ou **10e3** ou **5.88**

Tipos em VHDL

■ Integer:

- A um integer sempre é inferido um barramento de 32 bits a menos que a ele seja dado um range;
- É o tipo de dados que é utilizado pelo VHDL para indexar seus vetores (barramentos);
- O consumo de hardware é moderado.

■ Real:

- Possuem um alcance de 64 bits, porém mesmo definindo-se a ele um range ocupam o mesmo espaço em lógica, uma vez que parte dos bits é dada para a parte fracionária;
- Possui alto consumo de hardware, principalmente para as operações que o utilizam.

Tipos em VHDL

■ Enumeração:

- Cria uma lista dos possíveis valores que o identificador poderá assumir;
- Muito útil para criar um nível de abstração que facilite o projeto;
- É o tipo que permite a implementação de máquina de estados.

■ Físicos:

- São utilizados para representar grandezas físicas;
- Há um tipo físico já declarado nas bibliotecas VHDL, o tipo time;
- Estes tipo não são prototipáveis.

Tipos em VHDL

■ **Arrays:**

- São conjuntos de dados de um mesmo tipo;
- Geralmente são utilizados para representar espaços em memória.

```
type memoria is array (0 to 9) of std_logic_vector;
```

■ **Subtypes:**

- São declarados a partir de um tipo de dado já declarado;
- Não são muito usuais por sua aplicação ser replicada;

```
subtype natural is integer range 0 to integer'high;
```

Atributos

■ Funcionalidade:

- Os atributos exercem funções de análise de sinais;
- Exemplos:

event: retorna *true* se ocorreu uma transição de borda de *clock*, caso contrário retorna *false*.

active: retorna *true* se ocorreu uma transição em uma lógica, caso contrário retorna *false*.

last_event: retorna o tempo que demorou a transição do sinal;

last_active: retorna o tempo que o sinal demorou para realiza a transição lógica;

last_value: retorna o valor anterior a transição do sinal;

lenght: retorna o tamanho do *array* do sinal.

Comandos

■ *if*:

- É o comando em *VHDL*, juntamente ao *case* mais utilizado;
- Geralmente utilizado para limitar a ação de lógicas;
- Implementado somente de um *process*.

Sintaxe:

```
if Condição_1 then
    -- Lógica_1;
elsif Condição_2 then
    -- Lógica_2;
else
    -- Lógica_3;
end if;
```

Comandos

■ *if (generate):*

- Utilizado para criar caminhos de síntese;
- Facilita a parametrização de códigos em relação a *timing*, *área* ou *performance*;
- Implementado em qualquer ponto da *architecture*.

Sintaxe:

```
if Condição generate  
    -- Lógica_1;  
end generate;
```

Comandos

■ **case:**

- Geralmente utilizado para limitar a ação de lógicas;
- Muito utilizado para criar tabelas, conversores de barramento e máquinas de estado;
- Implementado somente dentro de um *process*.

Sintaxe:

```
case valor_comparado is  
  when valor_cte_compar_1 =>  
    -- Lógica_1;  
  when valor_cte_compar_2 =>  
    -- Lógica_2;  
  when valor_cte_compar_3 =>  
    -- Lógica_3;  
  when others =>  
    -- Lógica_padrão;  
end case;
```

Comandos

■ ***when*:**

- Pode ser utilizado fora de um process;
- Geralmente utilizado para *trigger* sinais em lógicas assíncronas.

Sintaxe:

```
sinal_saída <= '1' when Lógica else '0';
```

Comandos

■ *for (loop):*

- Pode ser utilizado tanto em lógica assíncrona com em síncrona;
- Não é prototipável.

Sintaxe:

```
label:  
for variável_de_indexação in limite_do_loop loop  
    -- lógica 1;  
    -- lógica 2;  
end loop label;
```

Comandos

■ ***for (generate):***

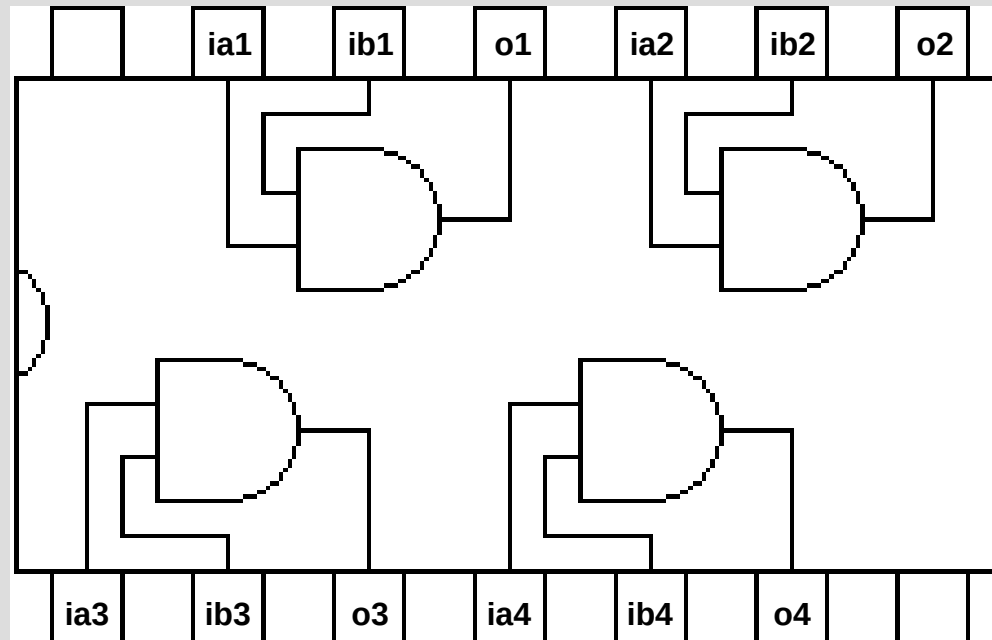
- Utilizado para facilitar a descrição de códigos em repetição;
- Não é prototipável.

Sintaxe:

```
label:  
for variável_de_indexação in limite_do_loop generate  
    -- lógica 1;  
    -- lógica 2;  
end generate label;
```

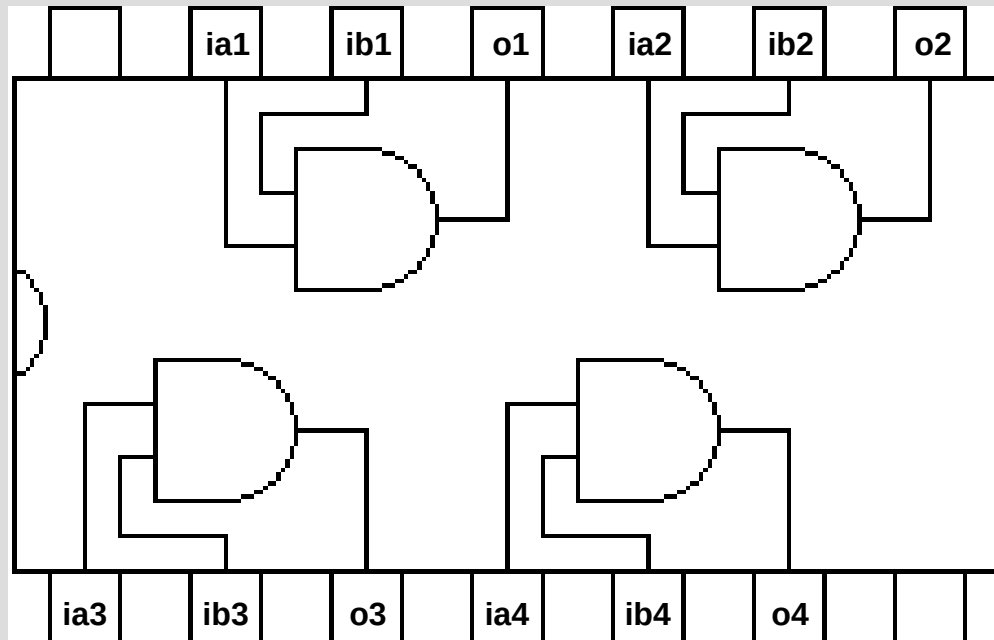
Comandos

- Implementação:



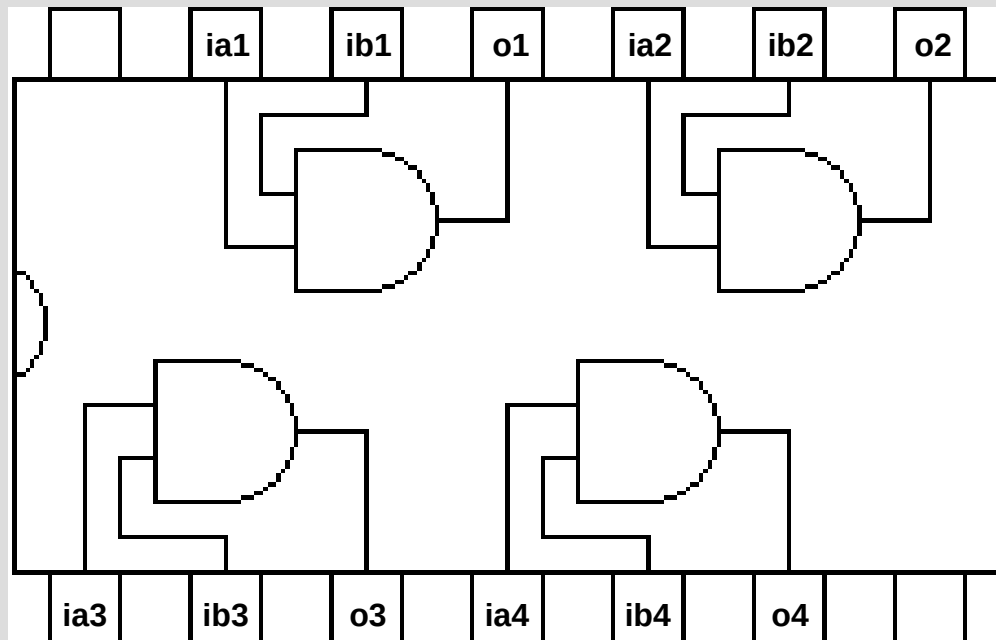
Comandos

- Implementação:



Comandos

■ Implementação:



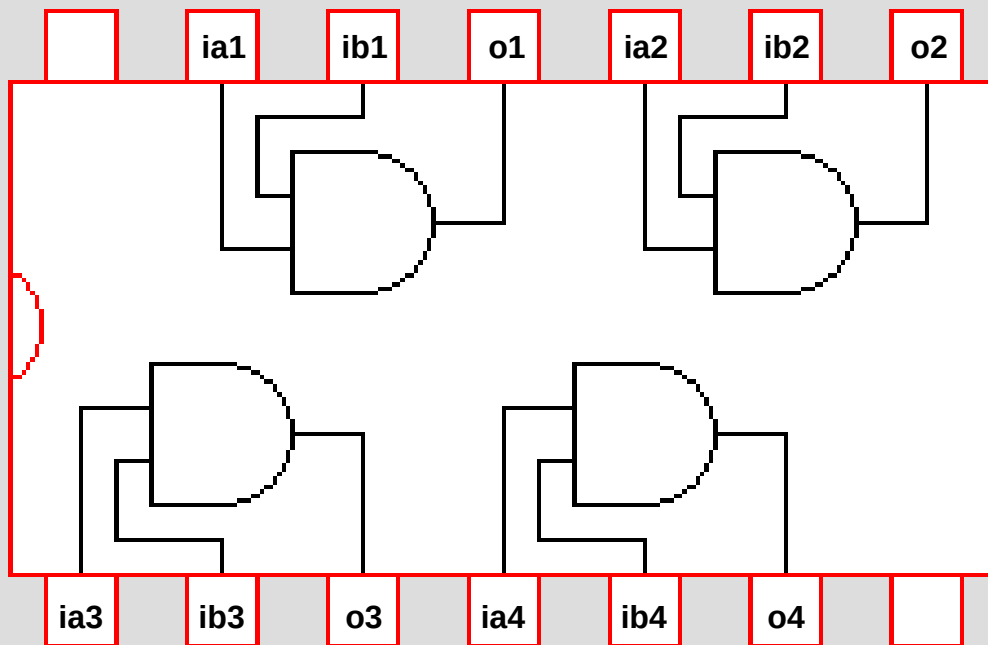
```
-- Bibliotecas
library ieee;
use ieee.std_logic_1164.all;

-- Entidade
entity CI7408 is
    port (
        ia : in  std_logic_vector(3 downto 0);
        ib : in  std_logic_vector(3 downto 0);
        o  : out std_logic_vector(3 downto 0)
    );
end CI7408;

-- Arquitetura
architecture CI7408 of CI7408 is
begin
    output_gen: for n in 0 to 3 generate
        o(n) <= ia(n) and ib(n);
    end generate;
end CI7408;
```

Comandos

■ Implementação:



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

-- Entidade

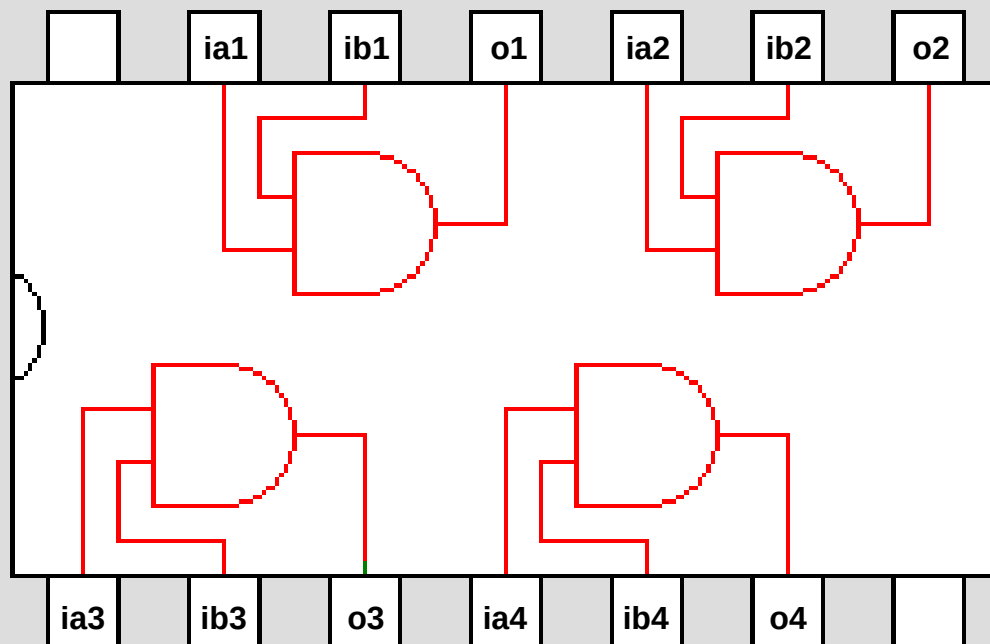
entity CI7408 is
  port (
    ia : in  std_logic_vector(3 downto 0);
    ib : in  std_logic_vector(3 downto 0);
    o  : out std_logic_vector(3 downto 0)
  );
end CI7408;

-- Arquitetura

architecture CI7408 of CI7408 is
begin
  output_gen: for n in 0 to 3 generate
    o(n) <= ia(n) and ib(n);
  end generate;
end CI7408;
```

Comandos

■ Implementação:



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

-- Entidade
entity CI7408 is
    port (
        ia : in  std_logic_vector(3 downto 0);
        ib : in  std_logic_vector(3 downto 0);
        o  : out std_logic_vector(3 downto 0)
    );
end CI7408;

-- Arquitetura
architecture CI7408 of CI7408 is
begin
    output_gen: for n in 0 to 3 generate
        o(n) <= ia(n) and ib(n);
    end generate;
end CI7408;
```

Comandos

■ ***while*:**

- Comando pouco utilizado por não ser prototipável e possuir um controle pouco eficiente;
- Geralmente utilizado em *testbenches*.

Sintaxe:

```
label:  
while expressão_booleana loop  
    -- lógica 1;  
    -- lógica 2;  
end loop label;
```

Comandos

■ ***wait***:

- Comando não prototipável;
- Utilizado somente em *testbenches*.

Sintaxe:

```
wait for tempo; (s, ms, ns, ...)  
wait on sinal;  
wait until condição;  
wait;
```

Implementação de Lógica Assíncrona

Implementações Assíncronas

■ Combinacional:

- Portas Lógicas;
- Comparadores;
- Interligações.

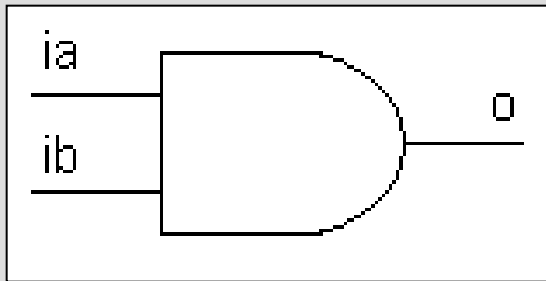
■ Computacional:

- *ULAs*;
- Conversores de barramento;
- Codificadores/Decodificadores;
- Multiplexadores/Demultiplexadores.

Implementações Assíncronas

- **Exemplo:**

Portas Lógicas:



```
o <= ia and ib;
```

ou

```
process(ia, ib  
begin  
    x <= ia and ib;  
end process;
```

Implementações Assíncronas

- Exemplo:

Código completo para uma porta *AND*:

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

-- Entidade
-----
entity and_port is
    port (
        -- Ports
        p0 : in    std_logic; -- Porta 0
        p1 : in    std_logic; -- Porta 1
        r  : out   std_logic;  -- Resultado
    );
end and_port;
-----

-- Arquitetura
-----
architecture and_port of and_port is
begin

    -----
    -- Lógica Assíncrona --
    -----

    r <= p0 and p1;
end and_port;
```

Implementações Assíncronas

- Exemplo:

Código completo para uma porta *AND*:

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

-- Entidade
-----
entity and_port is
  port (
    -- Ports
    p0 : in    std_logic; -- Porta 0
    p1 : in    std_logic; -- Porta 1
    r  : out   std_logic;  -- Resultado
  );
end and_port;
-----

-- Arquitetura
-----
architecture and_port of and_port is
begin
  -----
  -- Lógica Assíncrona --
  -----

  r <= p0 and p1;
end and_port;
```

Implementações Assíncronas

■ Exemplo:

Código completo para uma porta *AND*:

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

-- Entidade
-----
entity and_port is
port (
    -- Ports
    p0 : in    std_logic; -- Porta 0
    p1 : in    std_logic; -- Porta 1
    r  : out   std_logic;  -- Resultado
);
end and_port;

-- Arquitetura
-----
architecture and_port of and_port is
begin
    -----
    -- Lógica Assíncrona --
    -----

    r <= p0 and p1;
end and_port;
```

Bibliotecas utilizadas

Implementações Assíncronas

■ Exemplo:

Código completo para uma porta *AND*:

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

-- Entidade
entity and_port is
  port (
    -- Ports
    p0 : in    std_logic; -- Porta 0
    p1 : in    std_logic; -- Porta 1
    r  : out   std_logic;  -- Resultado
  );
end and_port;

-- Arquitetura
architecture and_port of and_port is
begin
  -----
  -- Lógica Assíncrona --
  -----

  r <= p0 and p1;
end and_port;
```

Bibliotecas utilizadas

Entidade (especificação das portas)

Implementações Assíncronas

■ Exemplo:

Código completo para uma porta *AND*:

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

-- Entidade
entity and_port is
  port (
    -- Ports
    p0 : in  std_logic; -- Porta 0
    p1 : in  std_logic; -- Porta 1
    r  : out std_logic  -- Resultado
  );
end and_port;

-- Arquitetura
architecture and_port of and_port is
begin
  -----
  -- Lógica Assíncrona --
  -----

  r <= p0 and p1;
end and_port;
```

Bibliotecas utilizadas

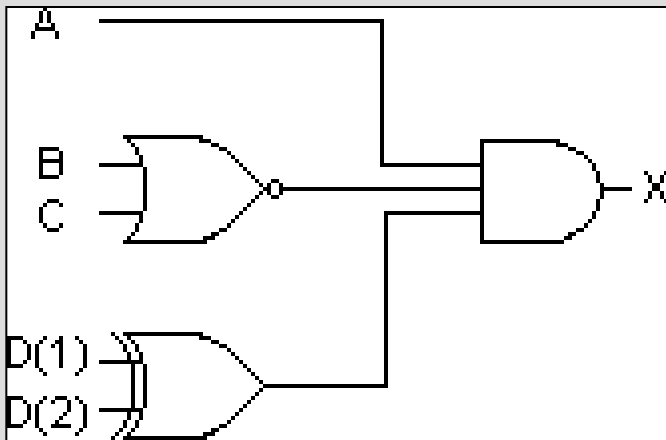
Entidade (especificação das portas)

Arquitetura (comportamento)

Implementações Assíncronas

■ Exemplo:

Portas Lógicas:



```
x <= A and not (B or C) and (D(1) xor D(2));
```

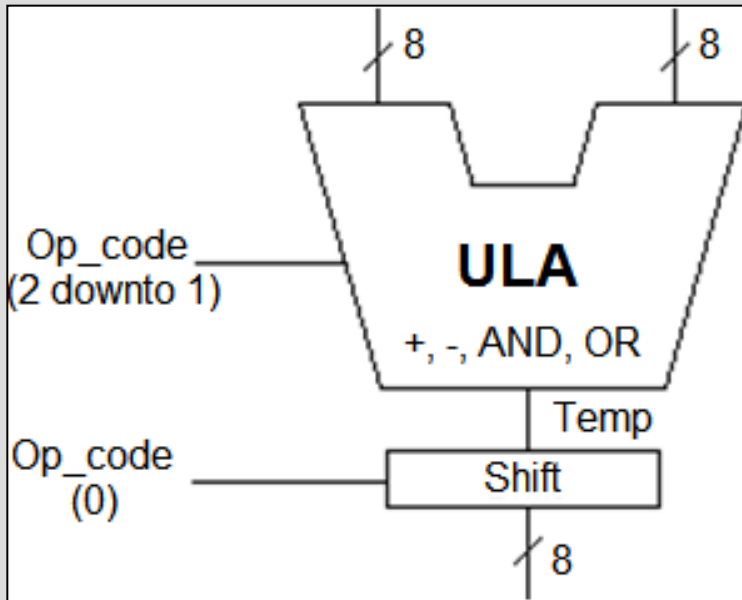
ou

```
process(A, B, C, D)
begin
    x <= A and not (B or C) and (D(1) xor D(2));
end process;
```

Implementações Assíncronas

■ Exemplo:

ULA (descrição):

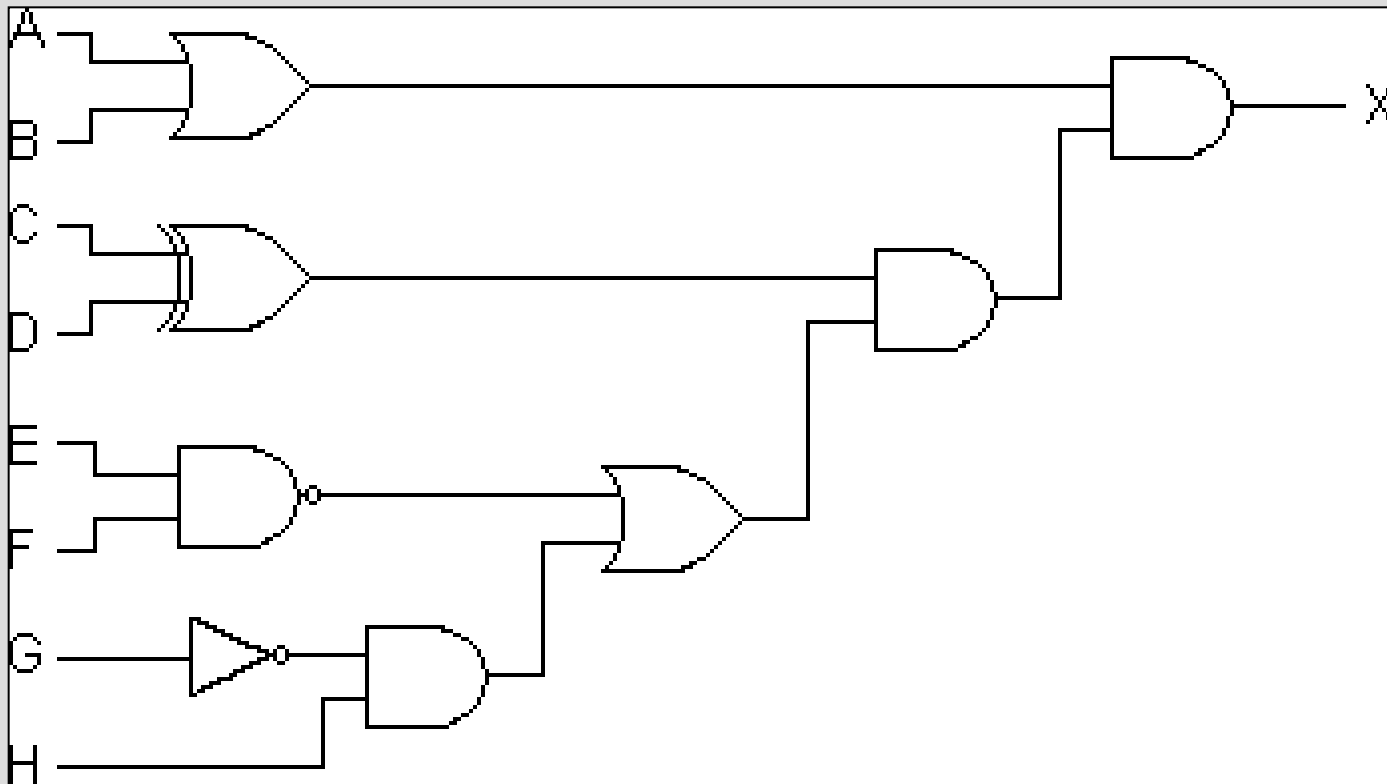


```
process(Op_code, Input_A, Input_B)
begin
  case Op_code(2 downto 1) is
    when "00" =>
      temp <= Input_A + Input_B;
    when "01" =>
      temp <= Input_A - Input_B;
    when "10" =>
      temp <= Input_A and Input_B;
    when "11" =>
      temp <= Input_A or Input_B;
    when others =>
      temp <= "00000000";
  end case;

  if Op_code(0) = '1' then
    ALU_output <= temp(6 downto 0) & '0';
  else
    ALU_output <= temp;
  end if;
end process ctrl;
```

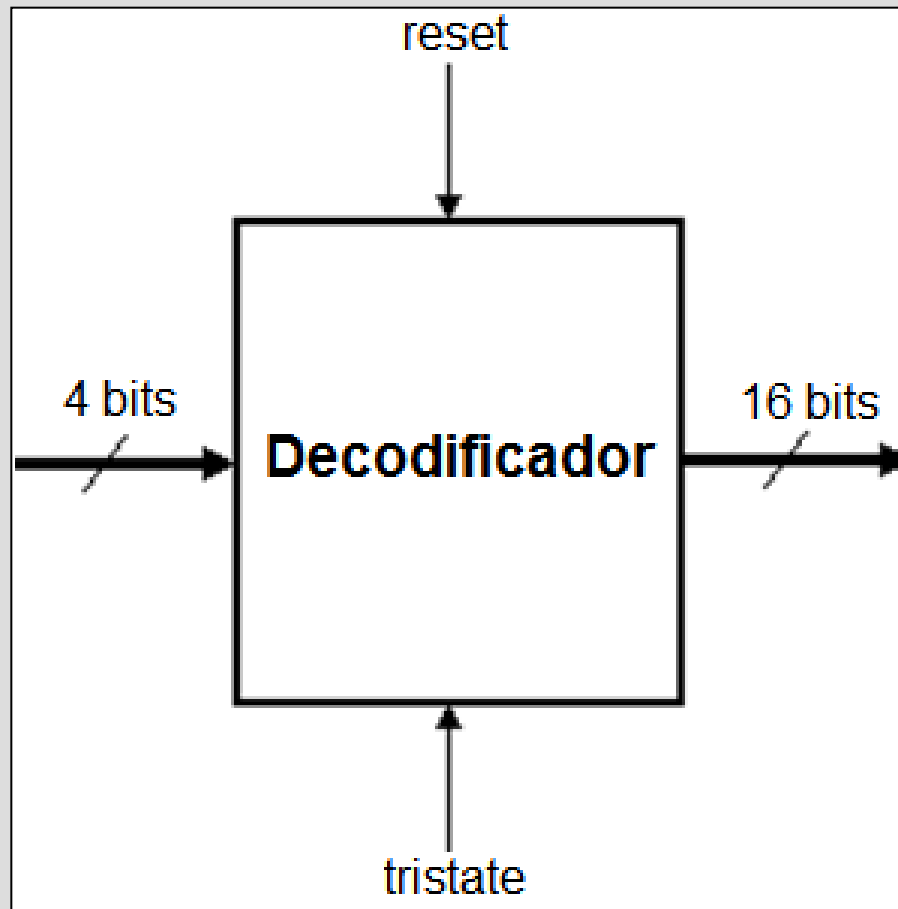
Exercícios

- Implemente o seguinte circuito combinacional:



Exercícios

- Implemente o seguinte decodificador:



Implementações Assíncronas

■ Exercício:

Crie módulos VHDL equivalentes as seguintes CIs:

- 7404: 6 (seis) portas NOT;
- 7408: 4 (quatro) portas AND;
- 7432: 4 (quatro) portas OR.

VHDL

ModelSim

Simulador Computacional

■ Apresentação:

- É um simulador computacional para análise de **sistemas digitais**;
- Possui alta fidelidade de resultados;
- Nas versões *full* inclusive é possível ter-se a simulação do tempo de porta das ligações dos circuitos lógicos;
- Nas versões *student* existem limitações, principalmente no que tange ao tempo de simulação, porém permanece uma poderosa ferramenta de simulação;
- Atualmente é o simulador de sistemas digitais mais aceito tanto pelo 'mundo acadêmico' como pela indústria.

■ Objetivos:

- Possuir fácil uso;
- Velocidade de simulação;
- Aprendizagem do usuário exponencial;
- Atende tanto ao usuário comum (projetos pequenos), como usuários avançados (grandes projetos)
- Oferecer suporte as mais modernas tecnologias (Sistemas Digitais);
- Suporte aos grandes fabricantes (Lógica Programável);
- Difusão no meio acadêmico (parcerias com projetos de pesquisa);
- Difusão no meio industrial (pacotes de *softwares*);
- Manuseio por IDE gráfica ou *scripts*.

■ Características:

- Capacidade de simular desde simples circuitos combinacionais à sistemas digitais complexos
- Possui uma ampla gama de configurações que podem ser personalizadas a qualquer tempo;
- *Frontend* feito totalmente em *Tcl/Tk*;
- *Middleend* criado em cima do *Tcl*;
- *Backend* totalmente implementado em Linguagem C;
- Alta performance de simulação (tanto qualitativa como quantitativa);
- Grande flexibilidade para automatização de processos (via *scripts*) e integração com outras ferramentas (*Gnuplot*, *Pearl*, *Lua*, *Python*, etc);
- Baseado em sistema UNIX (cuidado ao instalar com *Octave*, *SciLab*, *Cygwin*...)

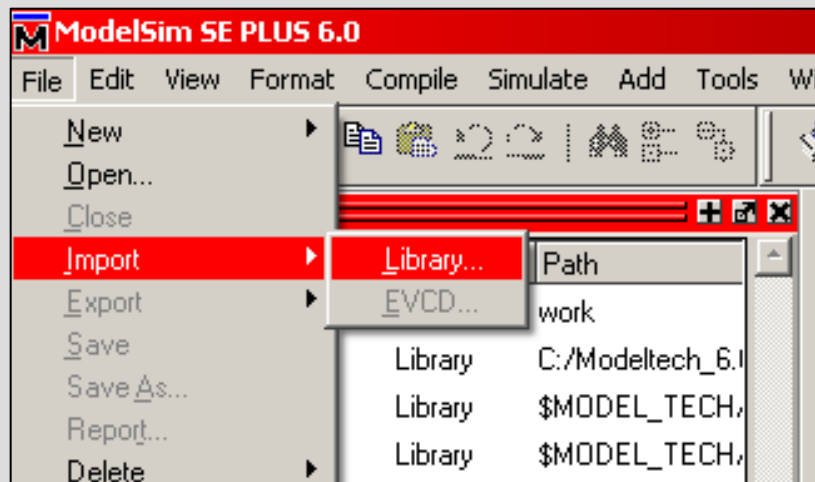
■ Instalação:

- Rodar o aplicativo “mti60se.exe”;
 - Escolher a opção “evaluation”.
- Copiar os arquivos *.zip para a pasta que o projeto foi criada;
 - Extrair o conteúdo das pastas
- Ordem de descompactação dos dados deve ser cuidada:
 - 1º: “mxe5.7g_6.2isp2_simulation_libraries.zip”;
 - 2º: “mxe5.7g_62i_ip1_1_xilinxcorelib.zip”.

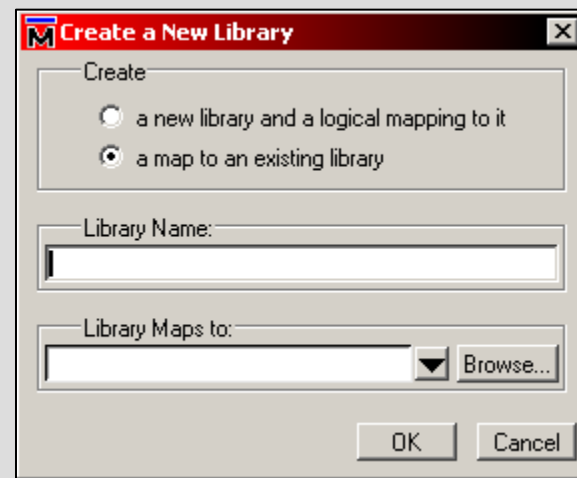
ModelSim

■ Configuração:

- Adição de *libraries*
- Com as *libraries* (diferencial do *ModelSim*) a resposta da simulação é melhorada;
- Para se adicionar uma *library* basta executar os seguintes passos:



File => Import =>
Library



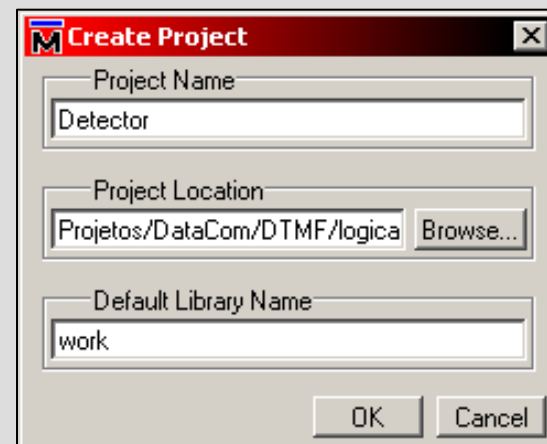
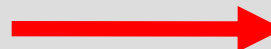
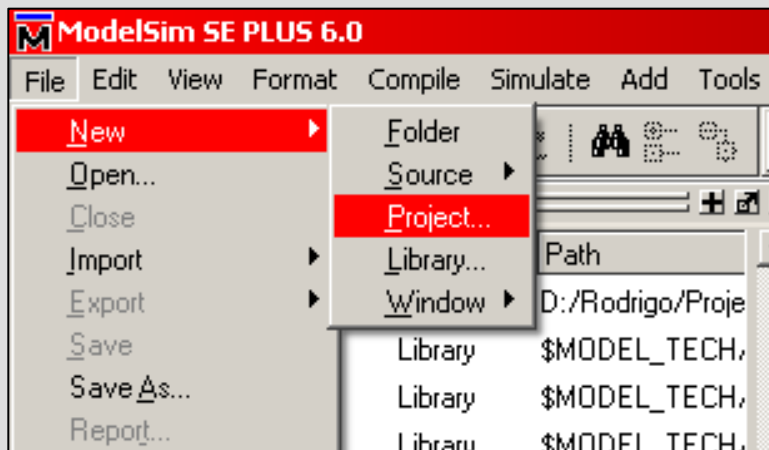
Informar o path da library
desejada

ModelSim

■ Funcionamento do Simulador:

• Fluxo de execução da simulação:

- Criação de um “projeto” (especifica configurações ao simulador);
- Compilação dos arquivos (são criadas bibliotecas de funções com o resultado da compilação);
- Início da simulação com a seleção dos sinais a serem monitorados;
- Rodar simulação (com estímulo via script ou lógica).

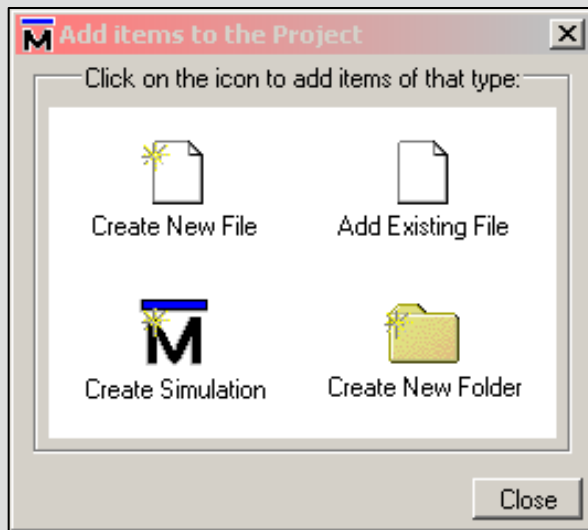


File => New => Project

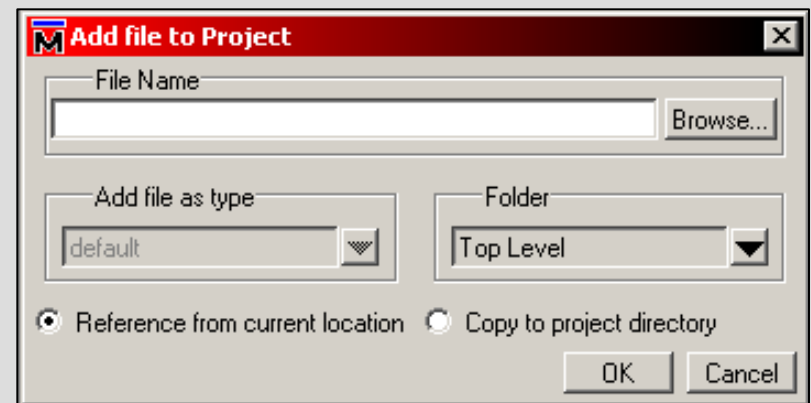
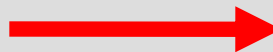
Nomear o projeto

■ Funcionamento do Simulador:

- **Fluxo de execução da simulação:**
 - Inserção de arquivos.



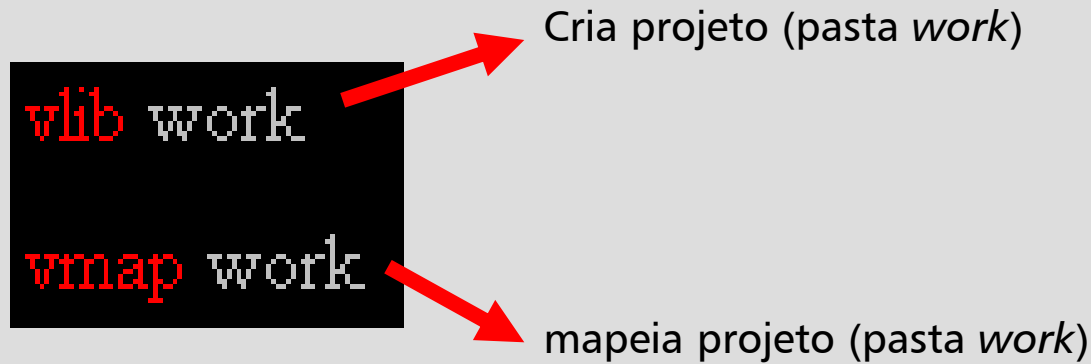
Escolher Add Existing File



Fornecer path do(s) arquivo(s)

■ Funcionamento do Simulador:

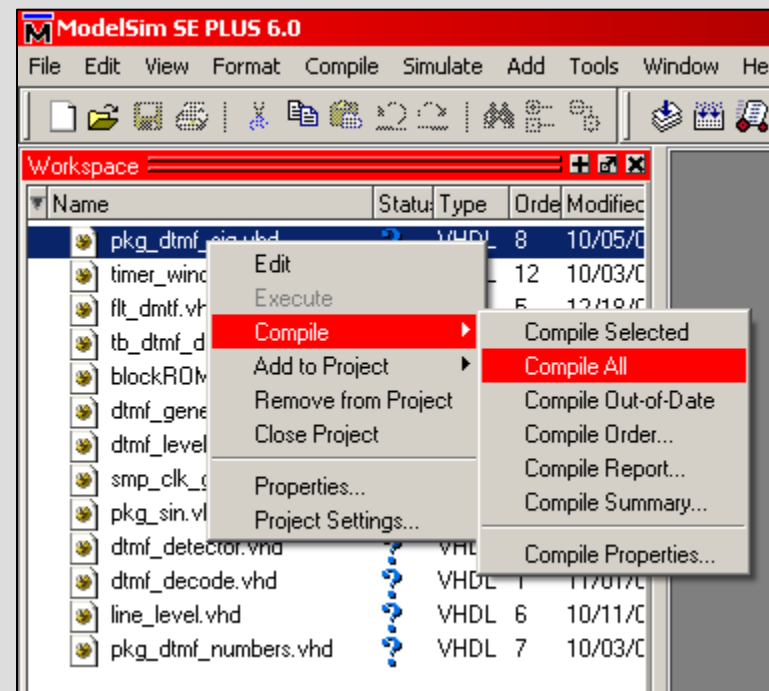
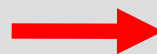
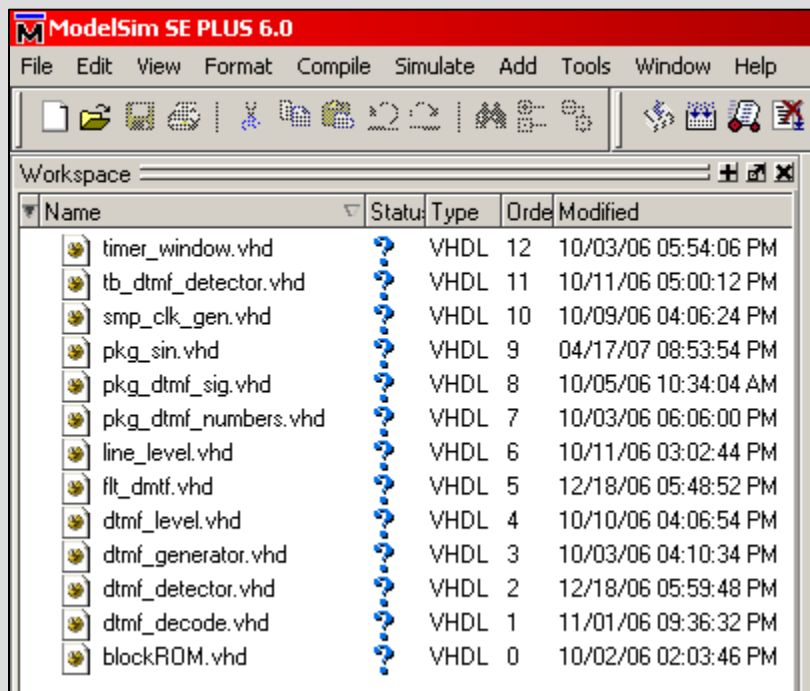
- **Fluxo de execução da simulação:**
 - O processo anterior pode ser substituído por duas linhas de comando.



ModelSim

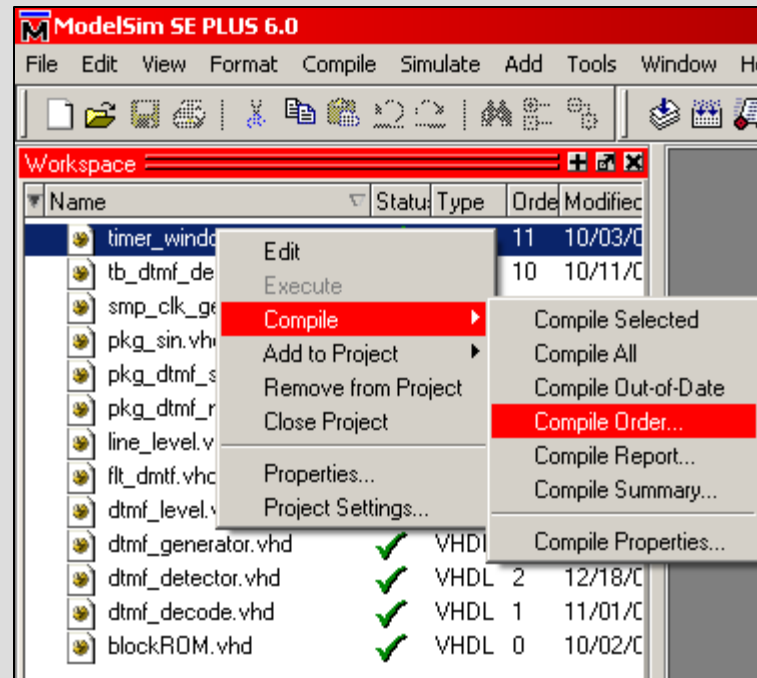
■ Funcionamento do Simulador:

- **Fluxo de execução da simulação:**
 - Compilação dos arquivos VHDL (compilação da lógica programável).



ModelSim

- **Funcionamento do Simulador:**
 - **Fluxo de execução da simulação:**
 - Compilação dos arquivos em ordem hierarquica.



Botão direito do mouse:

Compile => Compile Order

ModelSim

■ Funcionamento do Simulador:

- **Fluxo de execução da simulação:**
 - Compilação dos arquivos por *script TCL/ModelSim*.

```
vcom -93 -explicit -no1164 vhd/line_level.vhd
vcom -93 -explicit -no1164 vhd/timer_window.vhd
vcom -93 -explicit -no1164 vhd/blockROM.vhd
vcom -93 -explicit -no1164 vhd/flt_dtmf.vhd
vcom -93 -explicit -no1164 vhd/energia.vhd
vcom -93 -explicit -no1164 vhd/dtmf_level.vhd
vcom -93 -explicit -no1164 vhd/pkg_dtmf_numbers.vhd
vcom -93 -explicit -no1164 vhd/dtmf_decode.vhd
vcom -93 -explicit -no1164 vhd/mpu_simulator.vhd
vcom -93 -explicit -no1164 vhd/dtmf_detection.vhd
vcom -93 -explicit -no1164 vhd/pkg_dtmf_sig.vhd
vcom -93 -explicit -no1164 vhd/dtmf_generator.vhd
vcom -93 -explicit -no1164 vhd/tb_dtmf_detector.vhd
```

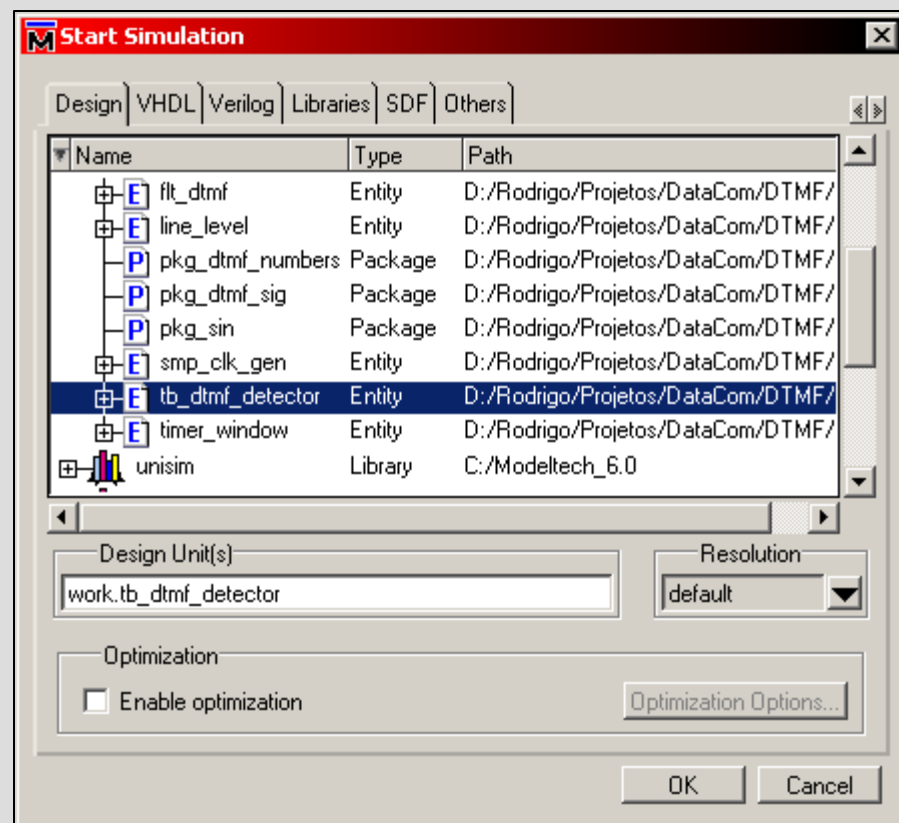
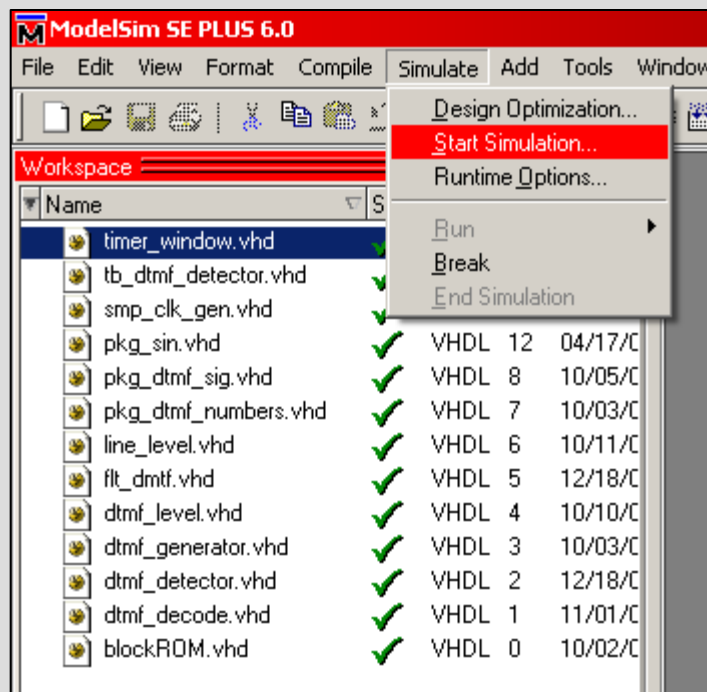
*Criar um arquivo *.do, fazendo sua chamada no prompt de comando*

Ex.: compile.do

ModelSim

■ Funcionamento do Simulador:

- **Fluxo de execução da simulação:**
 - Inicialização de uma simulação.



■ Funcionamento do Simulador:

- **Fluxo de execução da simulação:**
 - Inicialização de uma simulação via *scripts*.

```
vsim -xe tb_dtmf_detector -t ps
```

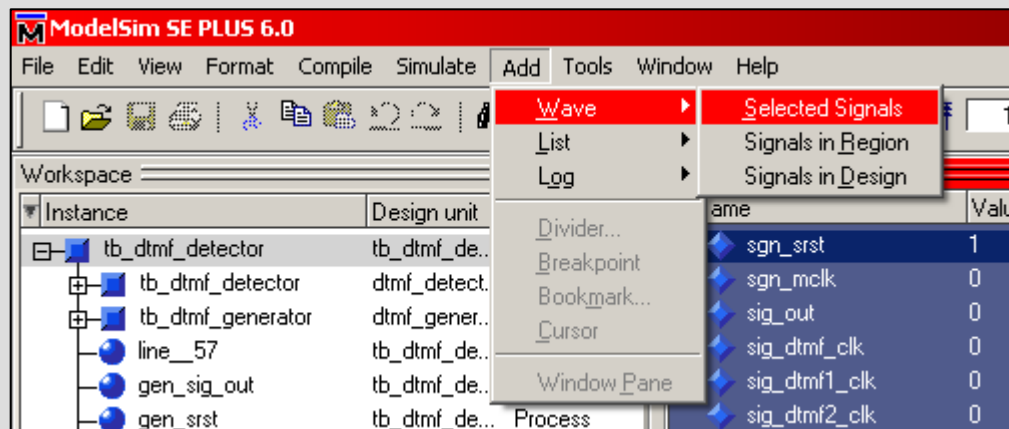
*Criar um arquivo *.do, fazendo sua chamada no prompt de comando*

Ex.: simula.do

ModelSim

■ Funcionamento do Simulador:

- **Fluxo de execução da simulação:**
 - Inicialização do visualizador de *waveforms*.

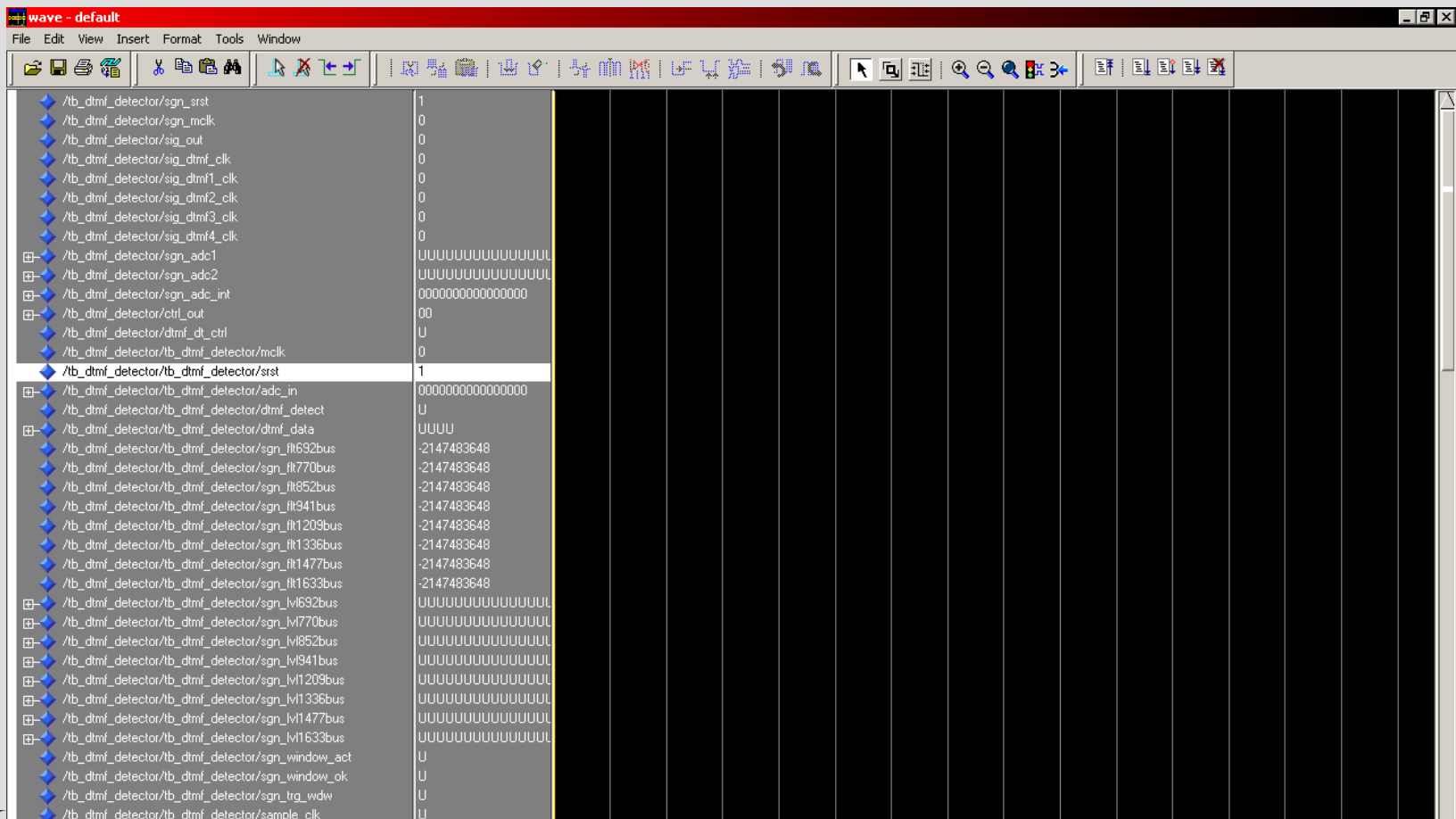


*Add => Wave => Select
Signals*

ou por linha de comando

*add wave -r /**

■ Funcionamento do Simulador:

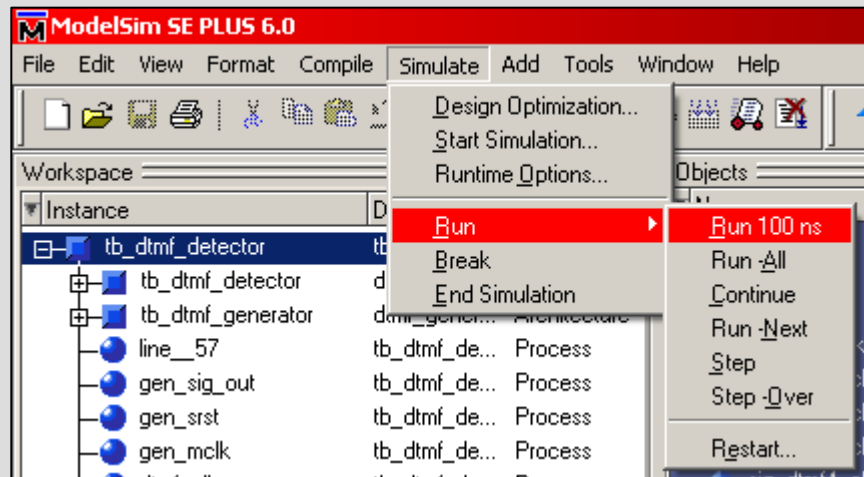


Exemplo de uma tela de waveform.

ModelSim

■ Funcionamento do Simulador:

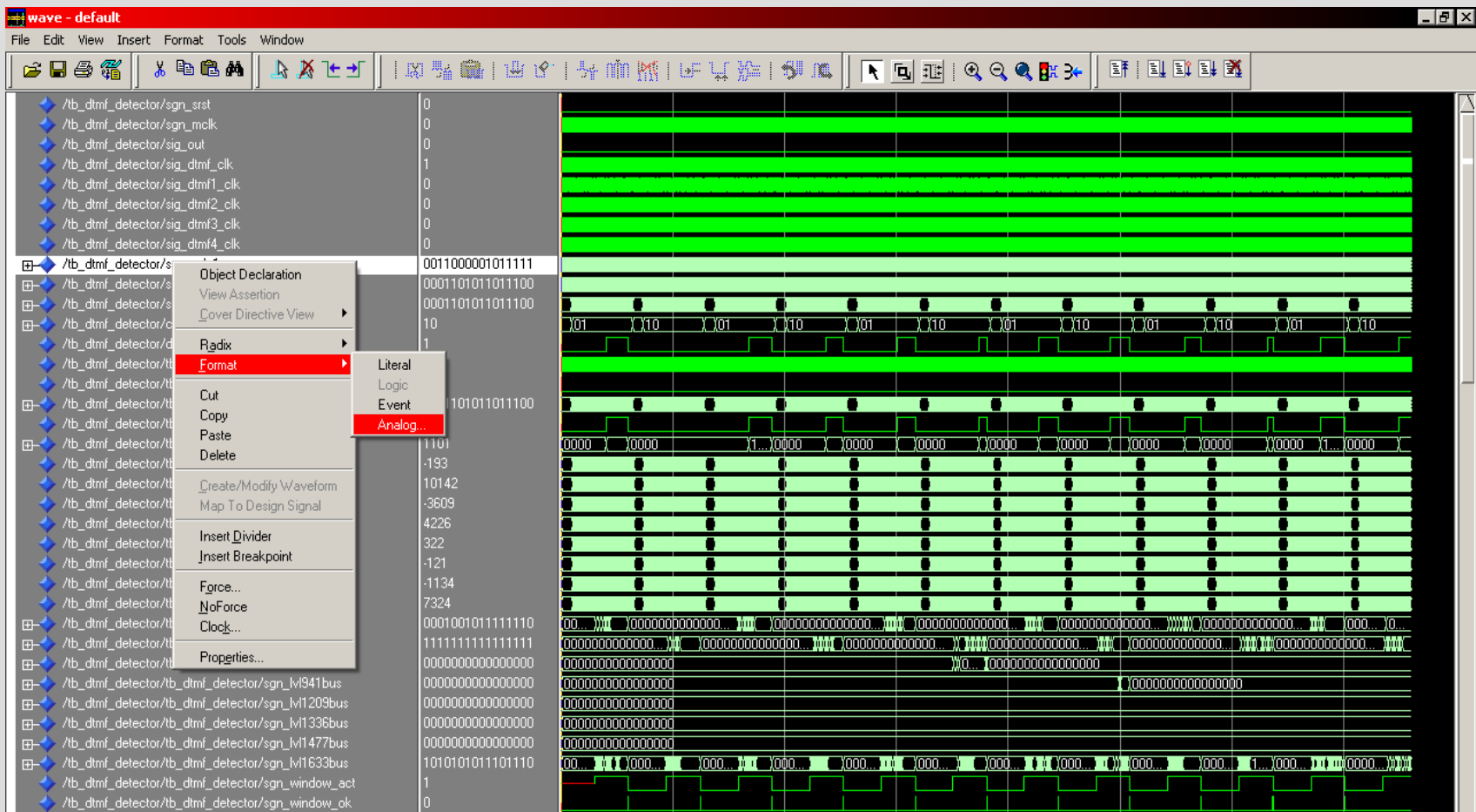
- **Fluxo de execução da simulação:**
 - Rodando a simulação.



Utilizando a IDE

Run => Opção desejada
ou por linha de comando
run tempo_desejado

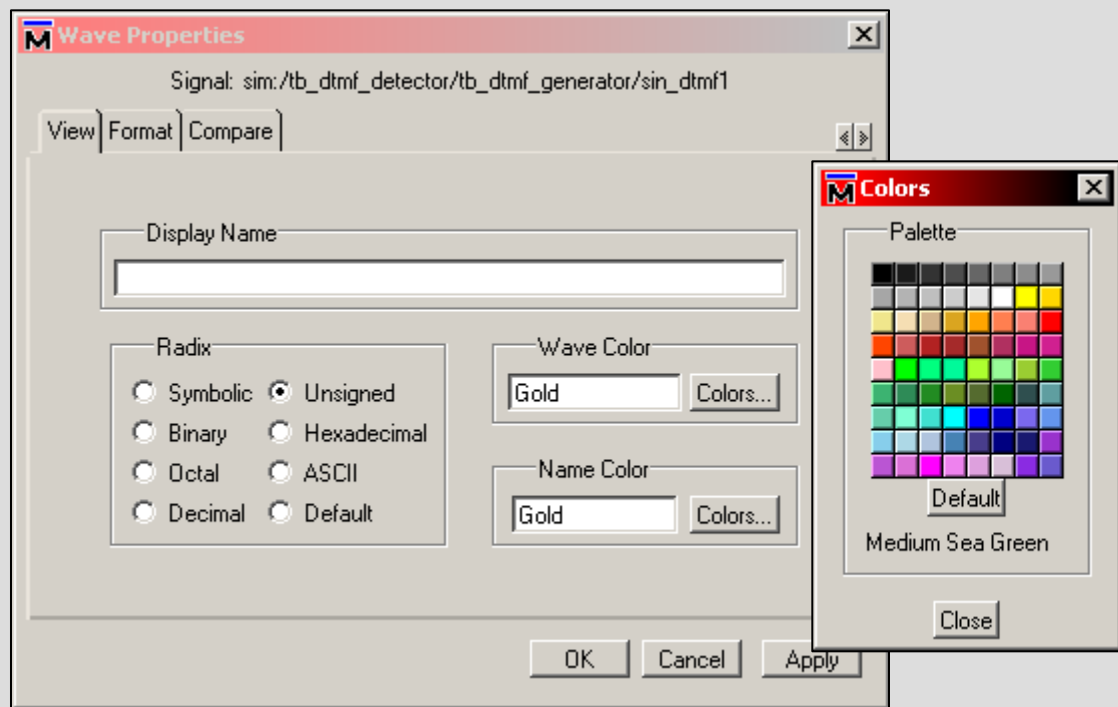
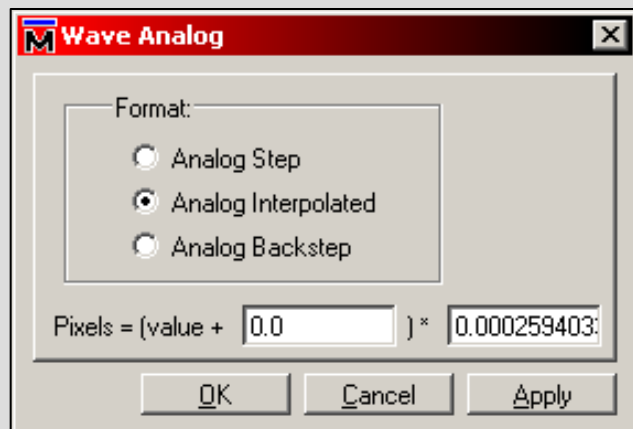
■ Funcionamento do Simulador:



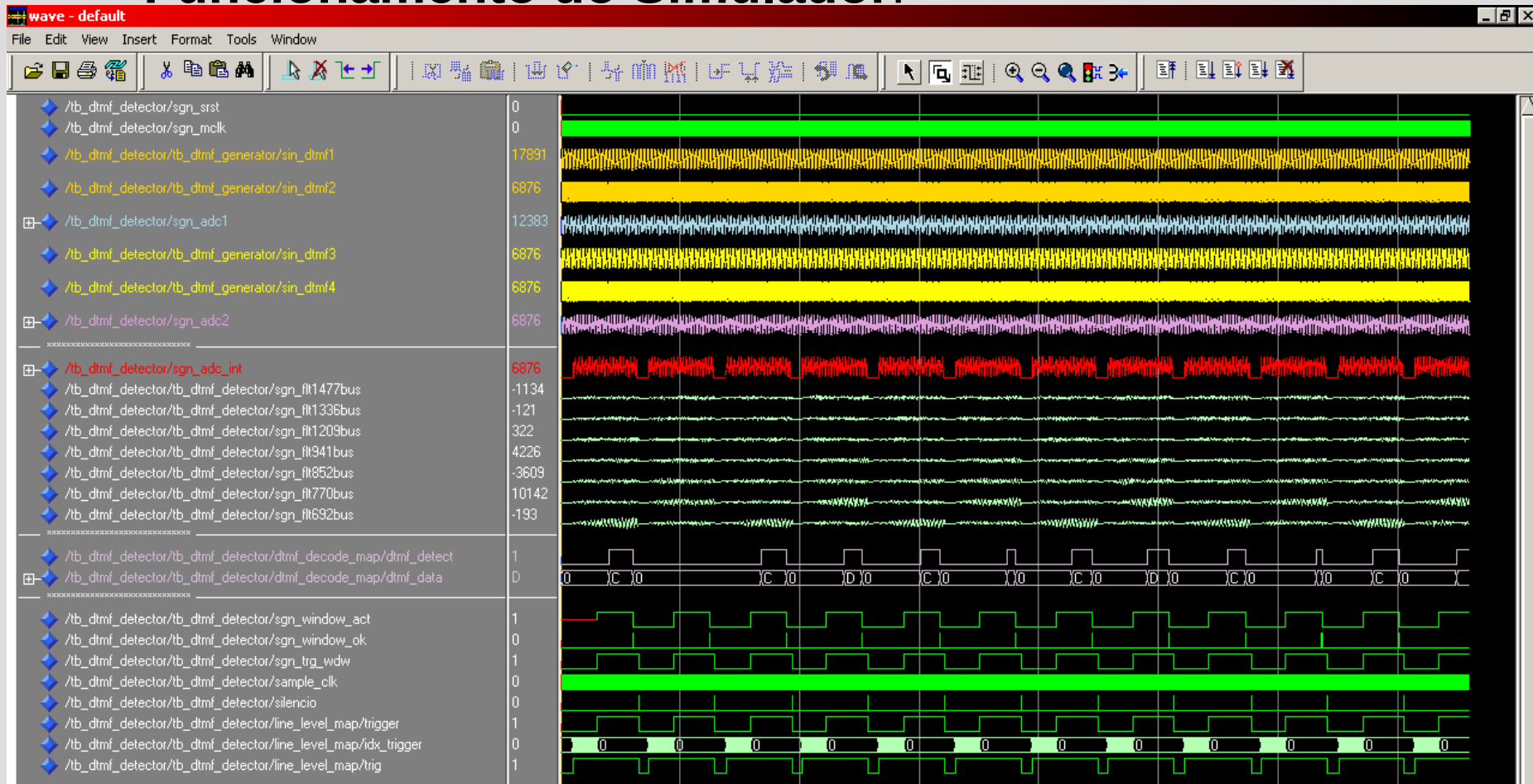
Resultado de uma simulação.

■ Funcionamento do Simulador:

- **Fluxo de execução da simulação:**
 - Melhorando a visualização das informações da simulação.

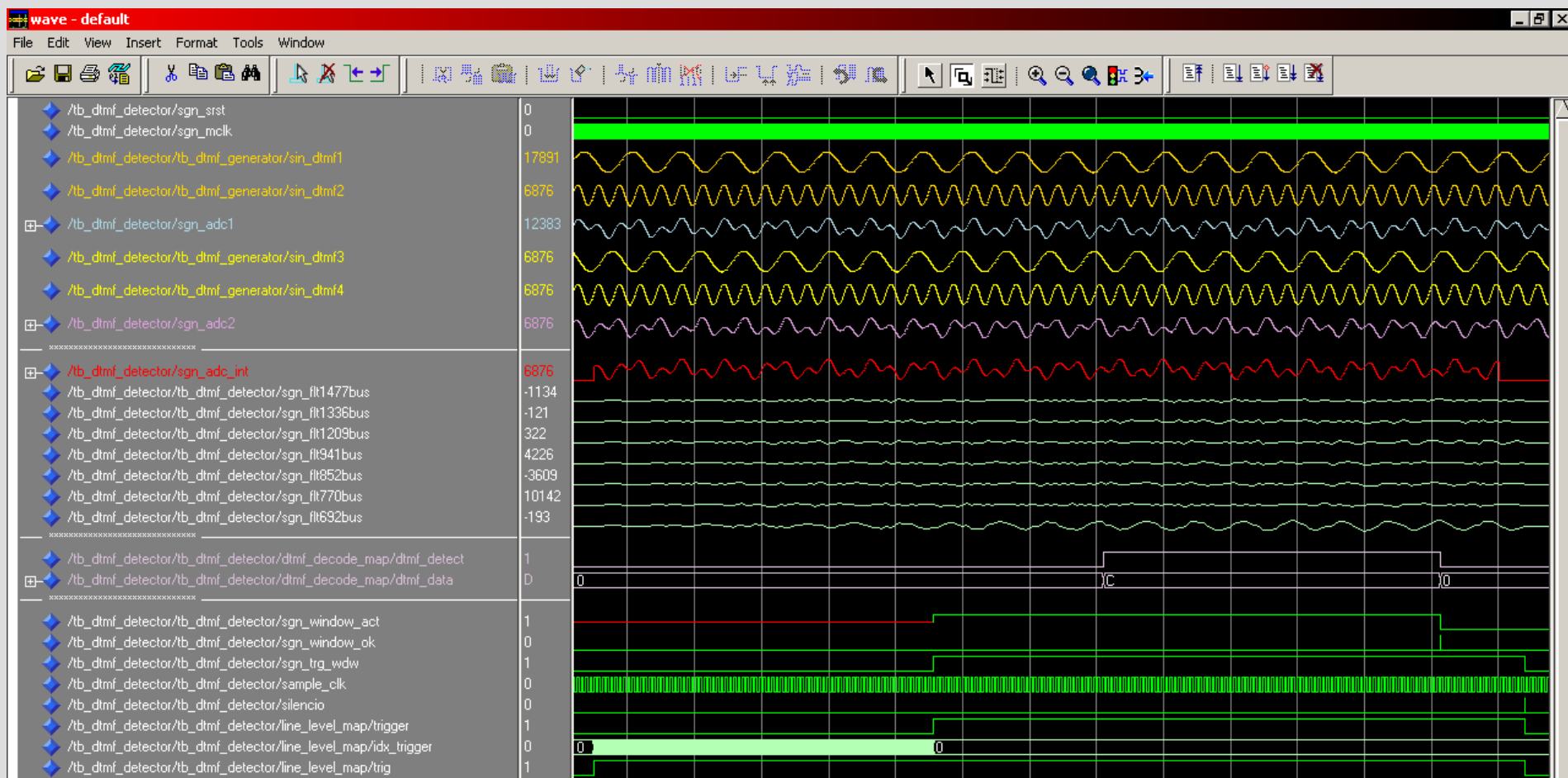


■ Funcionamento do Simulador:



Resultado de uma simulação.

■ Funcionamento do Simulador:



Resultado de uma simulação aplicando a ferramenta de zoom.

ModelSim

■ Funcionamento do Simulador:

- Automação dos processos:

- Todo o processo pode ser otimizado e passar a ser gerenciado em um único arquivo de *script* o qual faz as chamadas dos comandos *TCL/ModelSim*.

```
# Compila
    do c.do

# Simula
    vsim -xe tb_dtmf_detector -t ps

# Ondas
    do wave.do

    run 350 ms
```

VHDL

ISE Project Navigator

IDE para Prototipação de PLDs

ISE Project Navigator

■ Apresentação:

- O *ISE Project Navigator*, *Xilinx* é um conjunto de ferramentas dedicadas a todas as etapas de desenvolvimento com a tecnologia de Lógica Programável;
- Contemplas as seguintes etapas:
 - Síntese;
 - Roteamento;
 - *Placement*;
 - *Floorplanner*;
 - *Power Analyser*;
 - *FPGA Editor*;
 - *Core Generator*.

ISE Project Navigator

■ Características:

- Baseado totalmente em *scripts* integradores apresentados em uma *IDE* gráfica;
- Portanto é bastante fácil integrá-los a *scripts* proprietários aumentando a produtividade;
- Seus recursos são bastante confiáveis em termos de qualidade de resposta computacional;
- Computacionalmente possui 5 níveis de otimizadores desde a síntese, passando pela implementação até o *placement* final:
 - Redes Neurais Artificiais;
 - Algoritmo Bando de Pássaros;
 - Algoritmo Genético;
 - Algoritmo Caminho de Menor Custo;
 - Método de Otimização Determinístico (proprietário da *Xilinx*).

ISE Project Navigator

■ Exercício:

- Com base no tutorial implementar uma porta AND, onde as entradas serão dadas por botões e a saída é conectada a um LED.

```
library ieee;  
use ieee.std_logic_1164.all;  
  
entity and_port is  
  port (  
    i1 : in    std_logic;  
    i2 : in    std_logic;  
    o  : out   std_logic  
  );  
end and_port;  
  
architecture and_port of and_port is  
begin  
  o <= i1 and i2;  
end and_port;
```

Implementação de Lógica Síncrona

Implementações Síncronas

■ Combinacional:

- PLL's;
- Comparadores;
- Sincronizadores.

■ Computacional:

- *CPU's;*
- *Conversores de protocolo;*
- *DSP's (Filtros, espectrômetros, DFT's, DCT's, processamento de imagem, etc).*

Implementações Síncronas

■ Análise de bordas:

- Para a análise de bordas de sinais é utilizado o atributo *'event'*;
- Pode ser utilizado para analisar qualquer sinal do tipo *bit* ou *std_logic*;
- Estes sinais podem ser de entrada ou internos, nunca sinais de saída;
- O sintetizador irá sempre inferir um *flip-flop* para cada sinal de saída.

Implementações Síncronas

■ Análise de bordas:

- Para a análise de bordas de sinais é utilizado o atributo *'event'*;
- Pode ser utilizado para analisar qualquer sinal do tipo *bit* ou *std_logic*;
- Estes sinais podem ser de entrada ou internos, nunca sinais de saída;
- O sintetizador irá sempre inferir um *flip-flop* para cada sinal de saída.

■ Sintaxe:

```
if clk'event and clk = '1' then
```

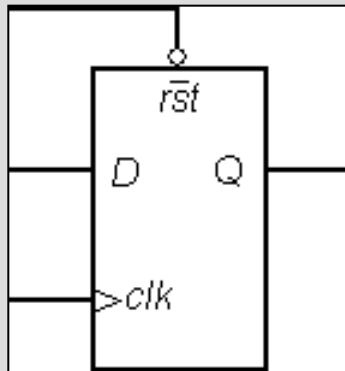
Observação:

Sempre que um sinal precisar ser registrado ele deverá passar por uma lógica que infira um flip-flop.

Implementações Síncronas

■ Exemplo:

- *Flip-Flop D*



Reset Assíncrono

```
process(nrst, clk)
begin
  if nrst = '0' then
    Q <= '0';

    elsif clk'event and clk = '1' then
      Q <= D;
    end if;
end process;
```

Reset Síncrono

```
process(clk)
begin
  if clk'event and clk = '1' then
    if nrst = '0' then
      Q <= '0';
    else
      Q <= D;
    end if;
  end if;
end process;
```

Implementações Assíncronas

■ Exemplo:

Código completo para uma porta *Flip-Flop D*:

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

-- Entidade
-----
entity flip_flop_d is
  port (
    -- Controle
    rst : in    std_logic; -- Reset (assíncrono)
    clk : in    std_logic; -- Clock
    -- Dados
    d   : in    std_logic; -- Entrada
    q   : out   std_logic; -- Saída
  );
end flip_flop_d;
-----

-- Arquitetura
-----
architecture flip_flop_d of flip_flop_d is
begin
  -----
  -- Processos --
  -----

  process(rst, clk)
  begin
    if rst = '0' then
      q <= '0';
    elsif clk'event and clk = '1' then
      q <= d;
    end if;
  end process;

end flip_flop_d;
```

Implementações Assíncronas

■ Exemplo:

Código completo para uma porta *Flip-Flop D*:

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

-- Entidade
-----
entity flip_flop_d is
  port (
    -- Controle
    rst : in  std_logic; -- Reset (assíncrono)
    clk : in  std_logic; -- Clock
    -- Dados
    d   : in  std_logic; -- Entrada
    q   : out std_logic; -- Saída
  );
end flip_flop_d;

-- Arquitetura
-----
architecture flip_flop_d of flip_flop_d is
begin
  -----
  -- Processos --
  -----

  process(rst, clk)
  begin
    if rst = '0' then
      q <= '0';
    elsif clk'event and clk = '1' then
      q <= d;
    end if;
  end process;

end flip_flop_d;
```

Bibliotecas utilizadas

Implementações Assíncronas

■ Exemplo:

Código completo para uma porta *Flip-Flop D*:

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

-- Entidade
entity flip_flop_d is
  port (
    -- Controle
    rst : in  std_logic; -- Reset (assíncrono)
    clk : in  std_logic; -- Clock
    -- Dados
    d   : in  std_logic; -- Entrada
    q   : out std_logic; -- Saída
  );
end flip_flop_d;

-- Arquitetura
architecture flip_flop_d of flip_flop_d is
begin
  -- Processos --
  process(rst, clk)
  begin
    if rst = '0' then
      q <= '0';
    elsif clk'event and clk = '1' then
      q <= d;
    end if;
  end process;
end flip_flop_d;
```

Bibliotecas utilizadas

Entidade (especificação das portas)

Implementações Assíncronas

■ Exemplo:

Código completo para uma porta *Flip-Flop D*:

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

-- Entidade
entity flip_flop_d is
  port (
    -- Controle
    rst : in  std_logic; -- Reset (assíncrono)
    clk : in  std_logic; -- Clock
    -- Dados
    d   : in  std_logic; -- Entrada
    q   : out std_logic; -- Saída
  );
end flip_flop_d;

-- Arquitetura
architecture flip_flop_d of flip_flop_d is
begin
  -- Processos --
  process(rst, clk)
  begin
    if rst = '0' then
      q <= '0';
    elsif clk'event and clk = '1' then
      q <= d;
    end if;
  end process;
end flip_flop_d;
```

Bibliotecas utilizadas

Entidade (especificação das portas)

Arquitetura (comportamento)

Implementações Assíncronas

■ Exemplo:

Código completo para uma porta *Flip-Flop D*:

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

-- Entidade
-----
entity flip_flop_d is
  port (
    -- Controle
    rst : in  std_logic; -- Reset (assíncrono)
    clk : in  std_logic; -- Clock
    -- Dados
    d   : in  std_logic; -- Entrada
    q   : out std_logic; -- Saída
  );
end flip_flop_d;

-- Arquitetura
-----
architecture flip_flop_d of flip_flop_d is
begin
  -- Processos --
  -----
  process(rst, clk)
  begin
    if rst = '0' then
      q <= '0';
    elsif clk'event and clk = '1' then
      q <= d;
    end if;
  end process;
end flip_flop_d;
```

Bibliotecas utilizadas

Entidade (especificação das portas)

Arquitetura (comportamento)

Implementações Assíncronas

■ Exemplo:

Código completo para uma porta *Flip-Flop D*:

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

-- Entidade
-----
entity flip_flop_d is
  port (
    -- Controle
    rst : in  std_logic; -- Reset (assíncrono)
    clk : in  std_logic; -- Clock
    -- Dados
    d   : in  std_logic; -- Entrada
    q   : out std_logic; -- Saída
  );
end flip_flop_d;
-----

-- Arquitetura
-----
architecture flip_flop_d of flip_flop_d is
begin
  -----
  -- Processos --
  -----

  process(rst, clk)
  begin
    if rst = '0' then
      q <= '0';
    elsif clk'event and clk = '1' then
      q <= d;
    end if;
  end process;

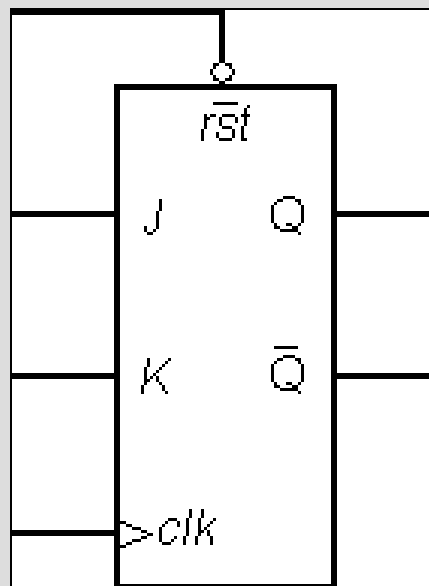
end flip_flop_d;
```

Processo

Implementações Síncronas

■ Exercício:

- Projete e simule um *Flip-Flop JK*

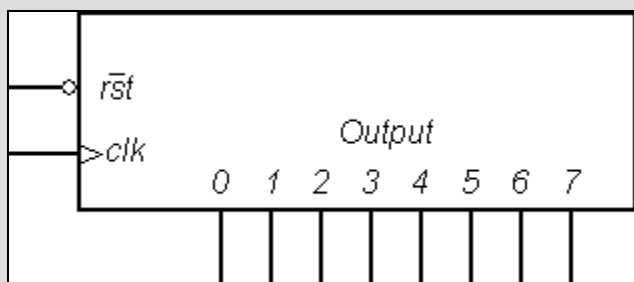


J	K	Q
0	0	mantém
0	1	0
1	0	1
1	1	comuta

Implementações Síncronas

■ Exemplo:

- Contadores



```
Output <= cont;
```

```
contador:
```

```
process(nrst, clk)
```

```
begin
```

```
if nrst = '0' then
```

```
    cont <= (others => '0');
```

```
elsif clk'event and clk = '1' then
```

```
    if cont < "11111111" then
```

```
        cont <= cont + '1';
```

```
    elsif cont = "11111111" then
```

```
        cont <= (others => '0');
```

```
    end if;
```

```
end if;
```

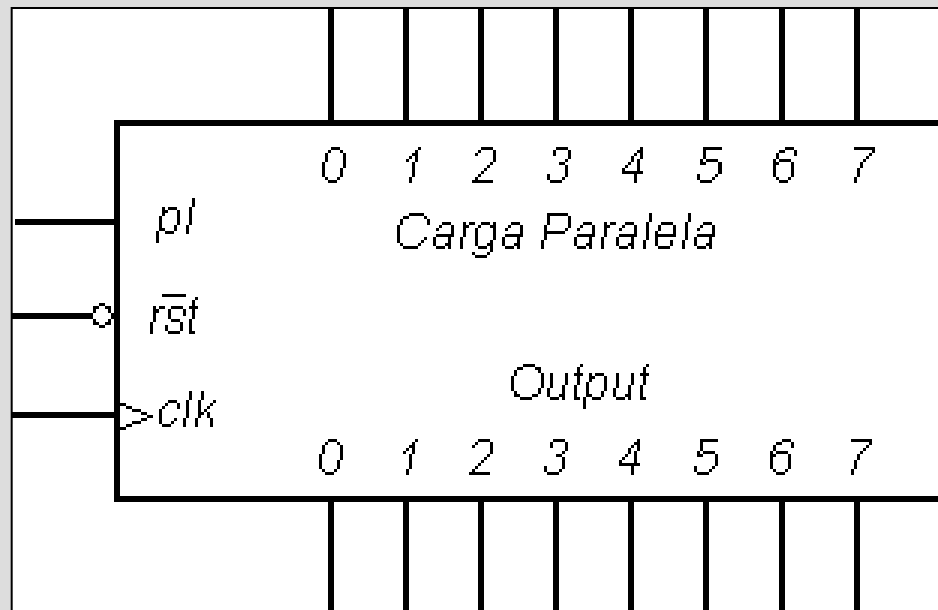
```
end process contador;
```



Implementações Síncronas

■ Exercício:

- Projete e simule um contador com carga paralela



Integração entre Módulos Descritos em VHDL

Integração de Módulos

■ **Motivação:**

• **Implementação de Projetos Modular:**

- Implementando projetos de maneira modular pode-se isolar erros mais facilmente ;
- Possibilita a divisão de tarefas entre projetistas em um mesmo projeto;
- Facilita documentação de manutenção e *upgrade*.

• **Reaproveitamento de códigos:**

- Códigos descritos anteriormente podem ser reaproveitados, simplesmente agregando-os ao projeto através de códigos de integração;
- Possibilidade de utilizar macros (instanciar) *hardwares* específicos de cada dispositivo;
- Capacidade de otimização de reaproveitamento de elementos de *hardware*.

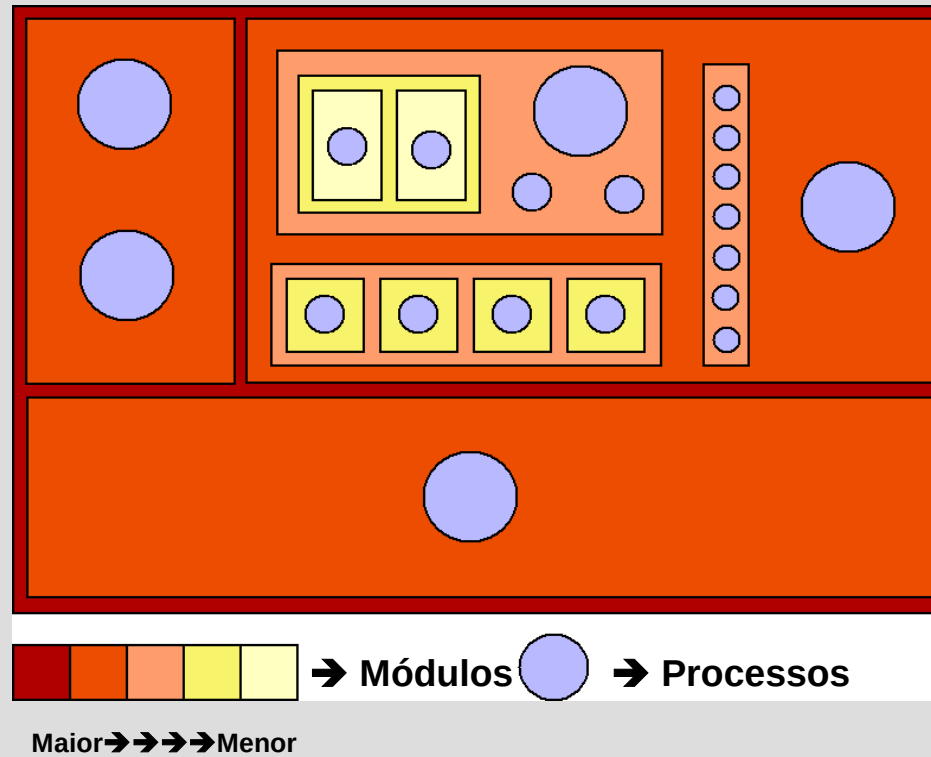
Integração de Módulos

■ Divisão em Níveis de Hierarquia:

- Todo projeto desenvolvido em *VHDL* pode possuir ***n*** níveis hierárquicos;
- Os códigos podem ser divididos em três níveis básicos de hierarquia entre si, inferior, equivalente e superior;
- Como convenção normalmente deixa-se o último nível (*Top-Level*) sem nenhum processo corrente, a menos que este seja um *Testbench*.

Integração de Módulos

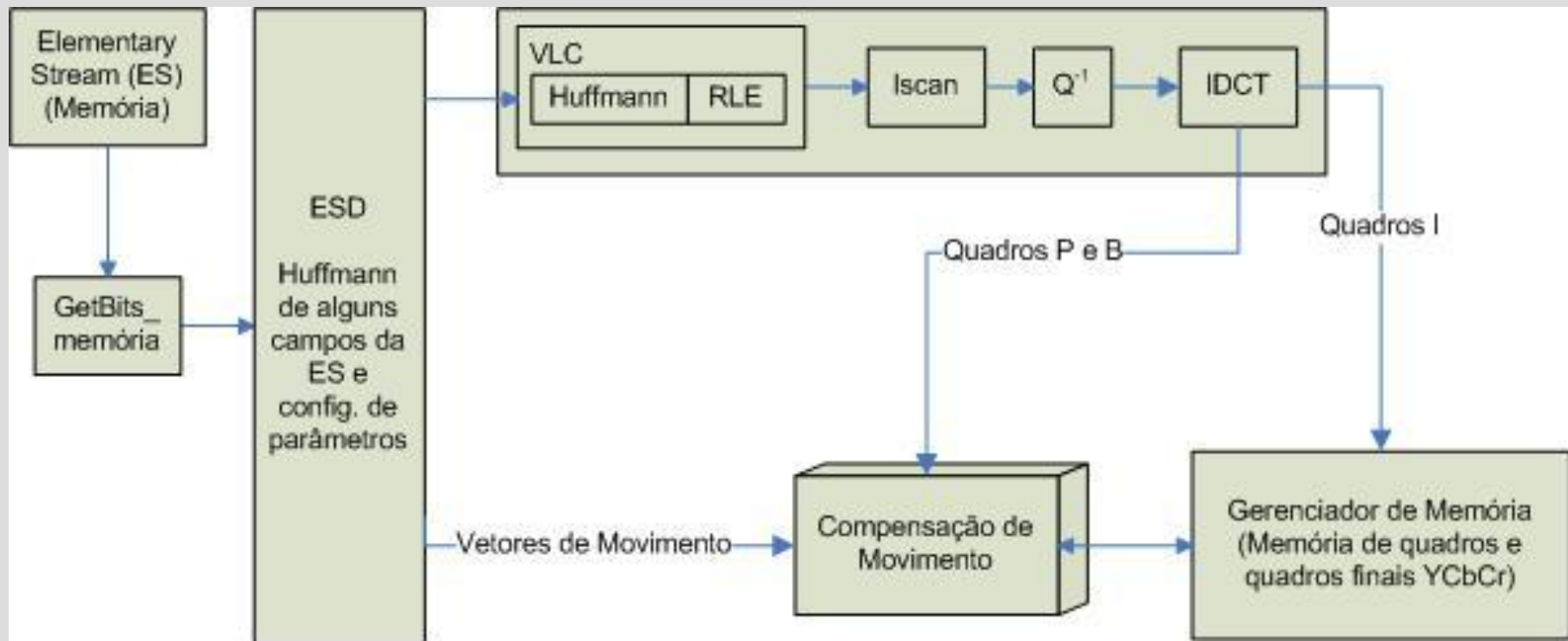
- Divisão em Níveis de Hierarquia:



Integração de Módulos

- **Divisão em Níveis de Hierarquia:**

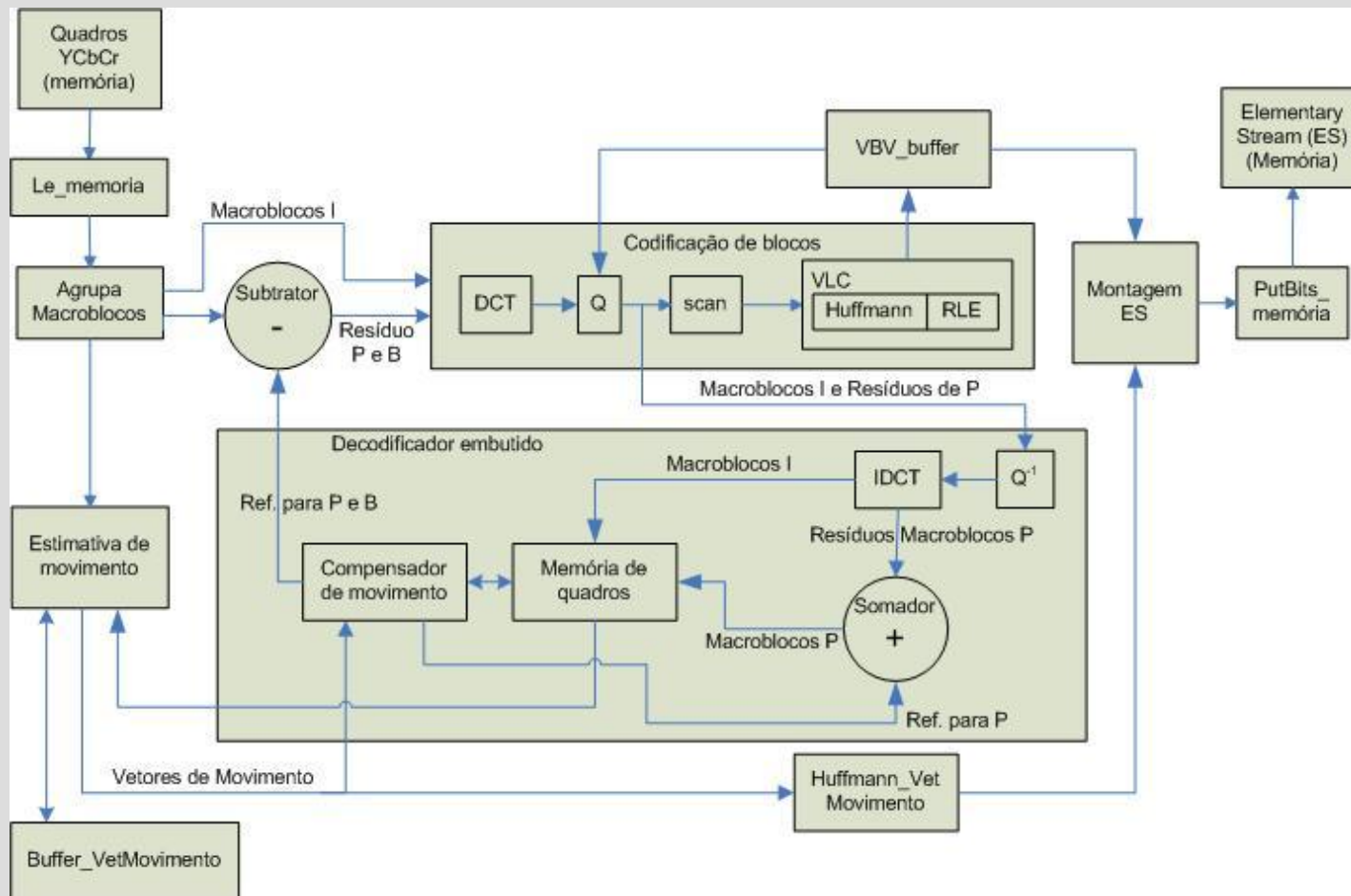
Implementação Prática:



Integração de Módulos

■ Divisão em Níveis de Hierarquia:

Implementação Prática:



Integração de Módulos

■ Declaração do *Top-Level*:

- A declaração do top-level é nada mais do que a declaração da entity do módulo:

```
-- libraries
entity nome_da_entidade is
    port(
        -- declaração das portas da integração
    );
end entity nome_da_entidade;
architecture nome_da_arquitetura of nome_da_entidade is
    -- declaração dos sinais (de controle e de integração)
    -- declaração dos componentes
begin
    -- port map dos módulos (componentes)
    -- processos
end architecture nome_da_arquitetura;
```

Integração de Módulos

■ Declaração de um *Component*:

- A declaração de um *component* nada mais é do que a descrição do protótipo do componente (módulo) *VHDL* descrito em outro arquivo;
- Este protótipo é a descrição de sua entidade.

```
component nome_do_arquivo is  
  port(  
    -- declaração das portas do componente  
    --Ex.:  
    rst : in    std_logic;  
    clk: in    std_logic;  
    q  : out std_logic  
  );  
end component nome_do_arquivo;
```

Integração de Módulos

■ Realização de um *Port Map*:

- Faz a ligação do *component* com o restante da descrição do *hardware*;
- *Port map* convencional (necessita da prévia declaração do *component*).

```
nome_da_integração : nome_do_arquivo
port map(
    -- lista das ligações com as portas do componente
    -- Ex:
    rst      => reset,
    clk      => clock,
    q        => output
);
```

Observação:

Note que o operador ' $\Rightarrow$ ' realiza o mapeamento do sinal e um *assignment*;

Note também que diferentemente da declaração das portas de um componente ou de uma entidade, é uma vírgula que separa as ligações dos sinais, sendo que o último não é separado por nenhum caractere.

Integração de Módulos

■ Realização de um *Port Map*:

- *Port map* otimizado (não necessita da prévia declaração do *component*).

```
nome_da_integração : entity work.nome_do_arquivo  
port map(  
    -- lista das ligações com as portas do componente  
    -- Ex:  
    rst      => reset,  
    clk      => clock,  
    q        => output  
);
```

Observação:

Com este tipo de declaração não existe a necessidade de se declarar o componente previamente.

FSM

Finite State Machine

Implementações Síncronas

■ Máquinas de Estado Finito (*FSM*):

- **Motivação:**

- Desenvolvida pela ***Teoria dos Autômatos***;
- Racionalizar sistemas complexos de decisão;
- Descrição de sistemas sequenciais.

- **Utilidade em Sistemas Digitais:**

- Permite simplificar lógicas de controle;
- Facilidade de codificação e implementação.

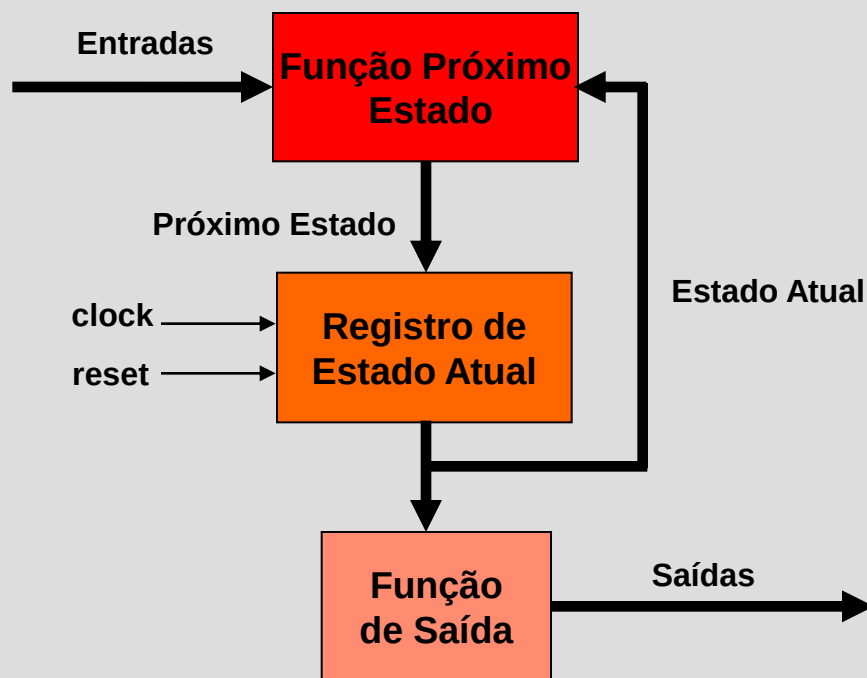
- **Tipos:**

- Máquina *Moore*;
- Máquina *Mealy*.

Implementações Síncronas

■ Máquinas de Estado Finito (*FSM*):

• Máquina *Moore*:



- Em uma máquina *Moore* a saída é função exclusiva do estado atual;

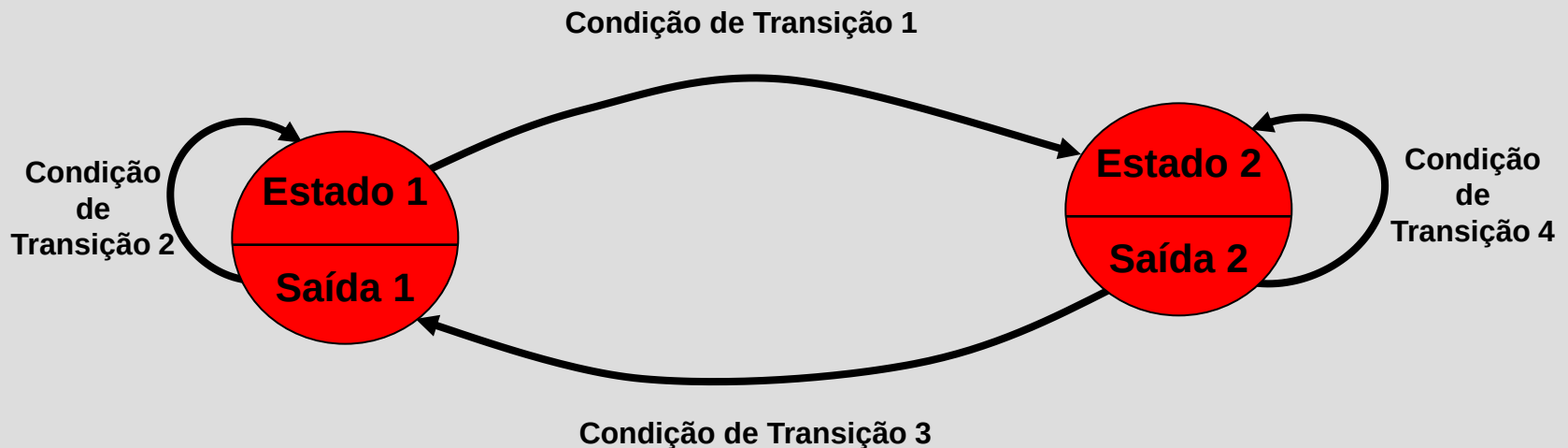
- É o tipo de máquina de estado mais comum e utilizada para implementar sincronismo e seqüencialidade de procedimentos;

- Geralmente é a máquina que implementa gerenciadores de transmissão e recepção de dados, ou analisadores de barramento.

Implementações Síncronas

- Máquinas de Estado Finito (*FSM*):

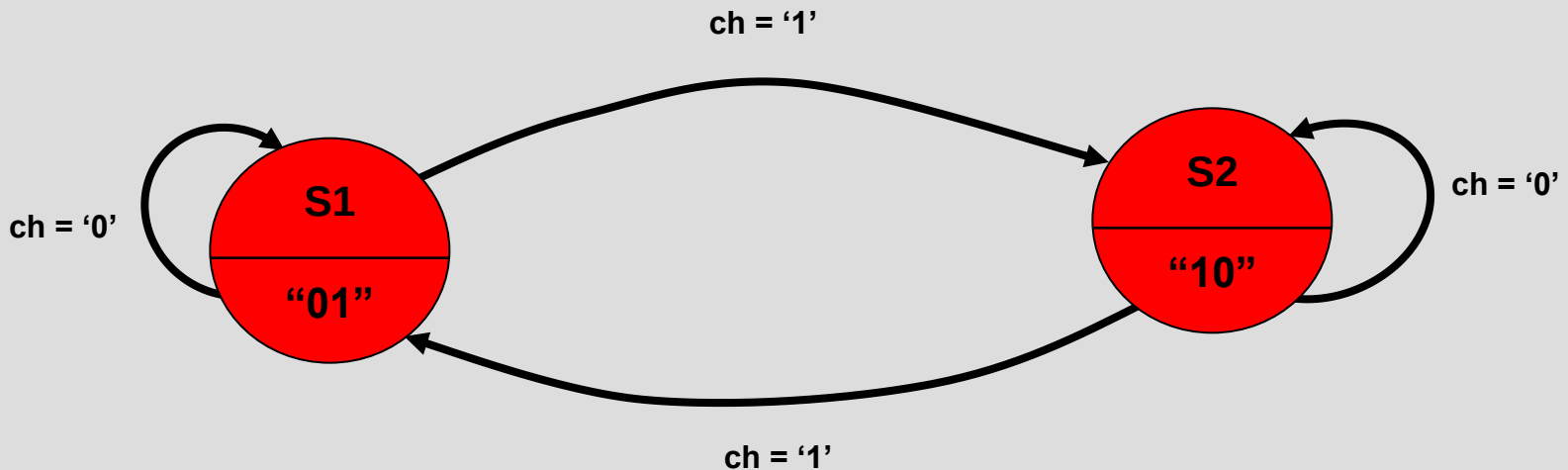
- Máquina *Moore*:



Implementações Síncronas

- Máquinas de Estado Finito (*FSM*):

- Máquina *Moore* (Exemplo):



Implementações Síncronas

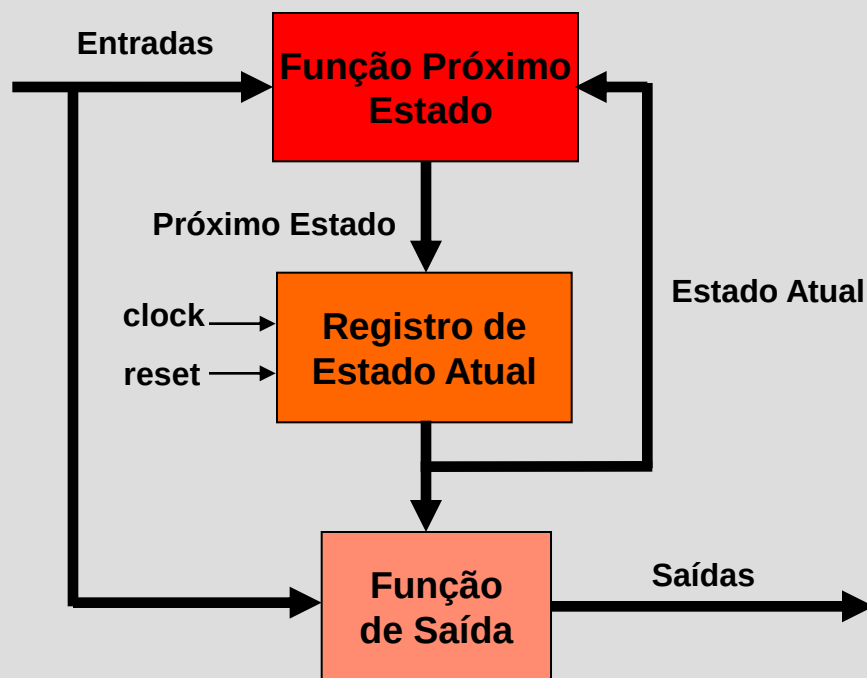
■ Máquinas de Estado Finito (*FSM*):

```
case estado is
  when S1 =>
    saida <= "01";
    if ch = '1' then
      estado <= S2;
    else
      estado <= S1;
    end if;
  when S2 =>
    saida <= "10";
    if ch = '1' then
      estado <= S1;
    else
      estado <= S2;
    end if;
  when others =>
    null;
end case;
```

Implementações Síncronas

- Máquinas de Estado Finito (*FSM*):

- Máquina *Mealy*:



- Em uma máquina *Mealy* a saída é função do estado atual e das entradas;

- É o tipo de máquina de estado pouco utilizada, porém muito útil para implementar operações assíncronas que são dependentes de seqüencialidade de procedimentos.

Implementações Síncronas

- Máquinas de Estado Finito (*FSM*):

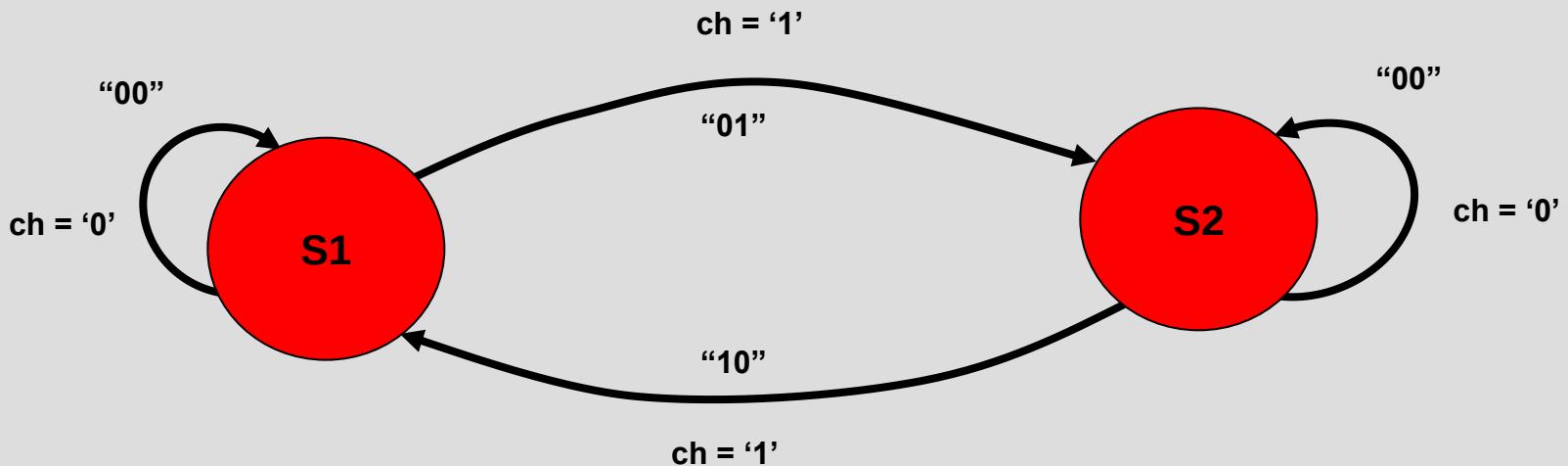
- Máquina *Mealy*:



Implementações Síncronas

- Máquinas de Estado Finito (*FSM*):

- Máquina *Mealy* (Exemplo):



Implementações Síncronas

■ Máquinas de Estado Finito (*FSM*):

```
case estado is
  when S1 =>
    if ch = '1' then
      estado <= S2;
    else
      estado <= S1;
    end if;
  when S2 =>
    if ch = '1' then
      estado <= S1;
    else
      estado <= S2;
    end if;
  when others =>
    null;
end case;
```

```
saida <= "01" when estado = S1 and entrada = '1' else
         "10" when estado = S2 and entrada = '1' else
         "00";
```

Implementações Síncronas

■ Máquinas de Estado Finito (FSM):

• Codificações de Estados:

- Codificação *One Hot*;
- Codificação *Gray*;
- Codificação Binária.

Estado	One Hot	Gray	Binário
S0	00000001	000	000
S1	00000010	001	001
S2	00000100	011	010
S3	00001000	010	011
S4	00010000	110	100
S5	00100000	111	101
S6	01000000	101	110
S7	10000000	100	111
Número total de transições	16	8	11
Máximo de transições por ciclo	2	1	3

Implementações Síncronas

■ Máquinas de Estado Finito (FSM):

- A implementação da codificação de estados pode ser realizada de duas maneiras básicas:
 - **Por enumeração***: onde o usuário apenas define os identificadores que corresponde a cada estado, sendo responsabilidade do sintetizador definir o tipo de codificação;
 - **Por definição**: neste modo o usuário define os valores assumidos por cada estado, e com isso acaba por definir a codificação desejada para a sua máquina de estados.

* Quando utilizado o modo de enumeração pode-se definir o tipo de codificação via passagem de parâmetros para o sintetizador, porém isto cria um *overhead* indesejável de síntese. Portanto quando se quer ter controle de qual codificação é utilizada recomenda-se fortemente utilizar-se o estilo de descrição '**por definição**'.

Implementações Síncronas

- Máquinas de Estado Finito (*FSM*):

- Exemplo de descrição utilizando o estilo de **enumeração**:

```
.  
.   
.   
architecture fsm of fsm is  
  type tipo_estado is (S0, S1);  
  
  signal estado : tipo_estado;  
begin  
  .  
  .  
  .  
end fsm;
```

Implementações Síncronas

■ Máquinas de Estado Finito (*FSM*):

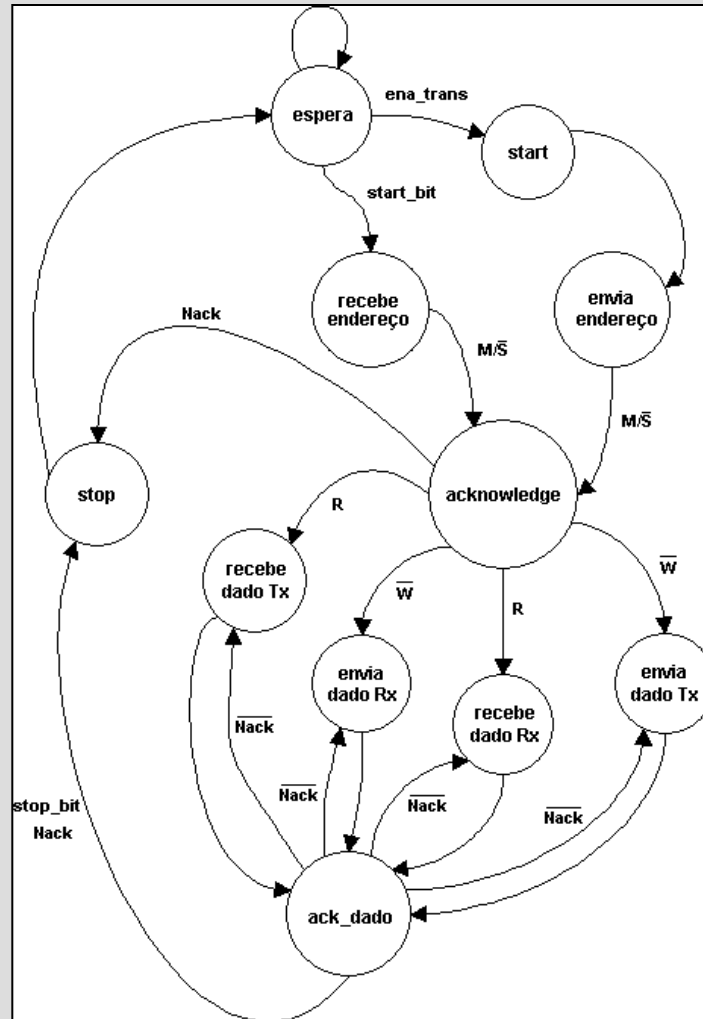
- Exemplo de descrição utilizando o estilo de **definição**:

```
.  
.   
.   
architecture fsm of fsm is  
    constant S0 : std_logic_vector(2 downto 0);  
    constant S1 : std_logic_vector(2 downto 0);  
  
    signal estado : std_logic_vector(2 downto 0);  
  
begin  
    .  
    .  
    .  
end fsm;
```

Implementações Síncronas

■ Máquinas de Estado Finito (FSM):

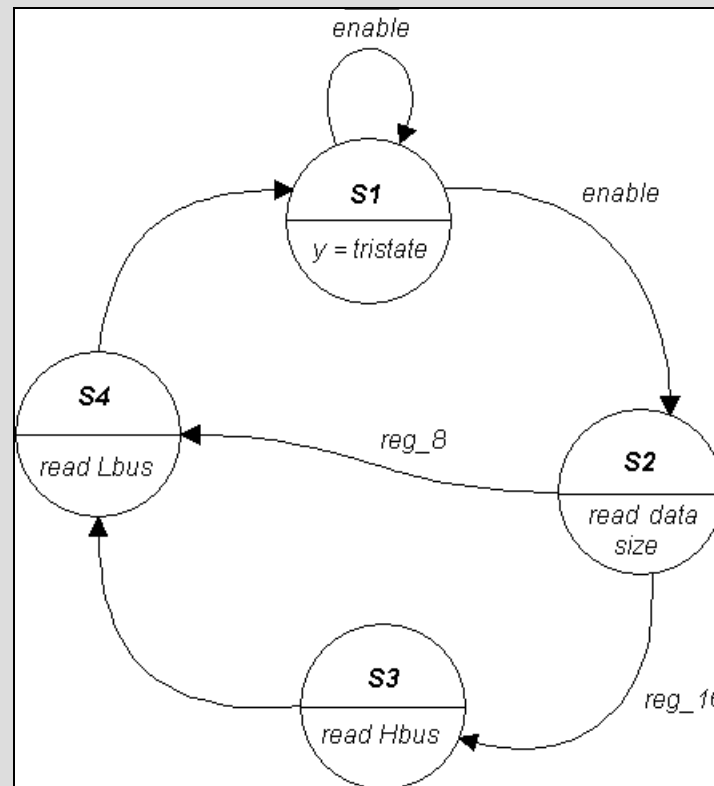
• Exemplos:



Implementações Síncronas

- Máquinas de Estado Finito (*FSM*):

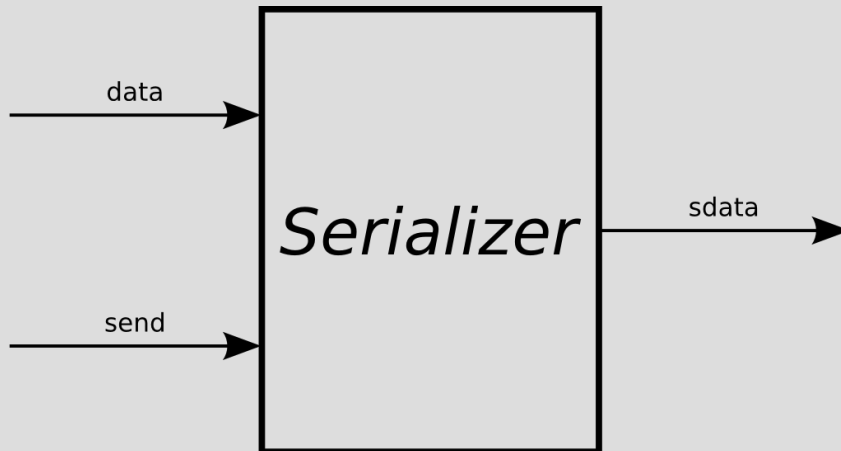
- Exemplos:



Implementações Síncronas

- Máquinas de Estado Finito (*FSM*):

- Exercício:

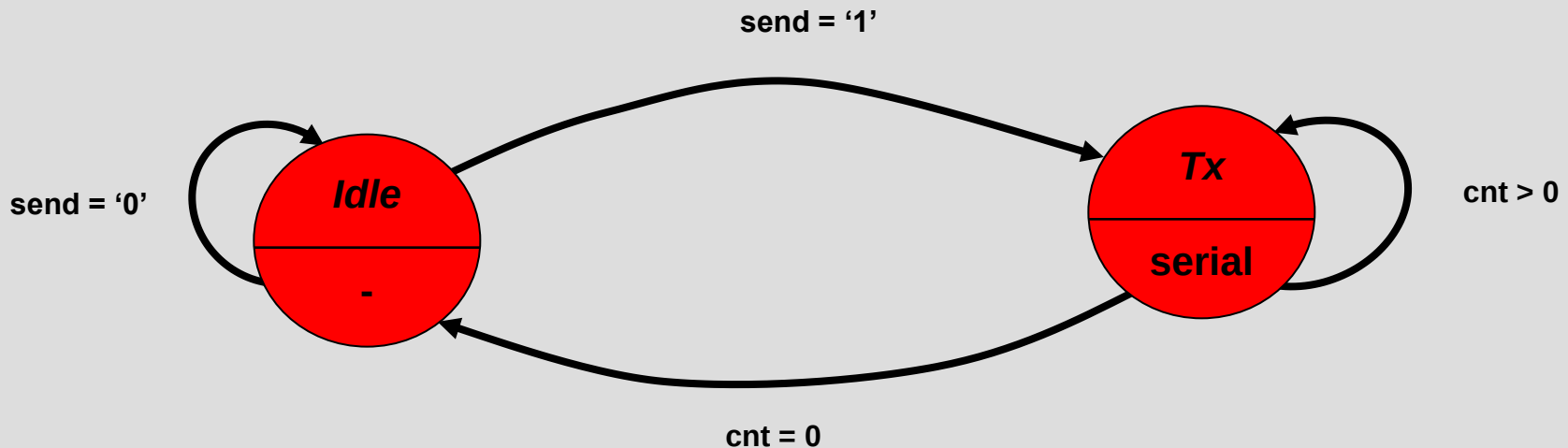


Implemente e simule um módulo *serializer* em *VHDL*.

Implementações Síncronas

- Máquinas de Estado Finito (*FSM*):

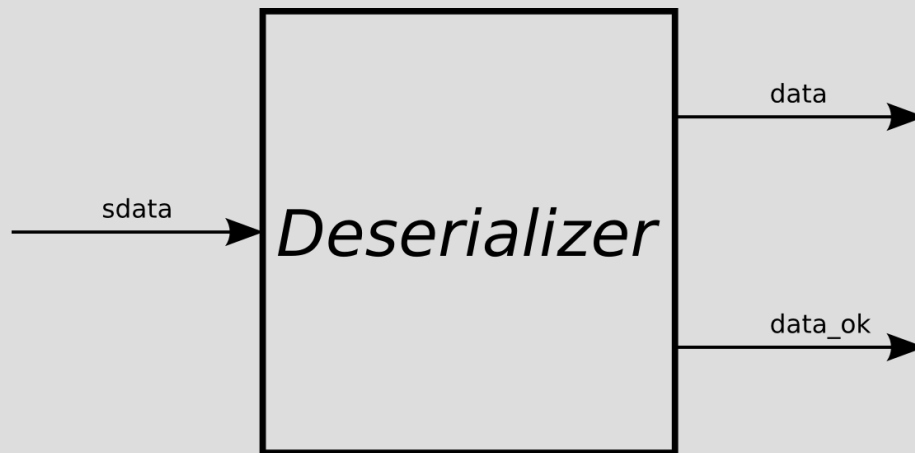
- Exercício:



Implementações Síncronas

- Máquinas de Estado Finito (*FSM*):

- Exercício:

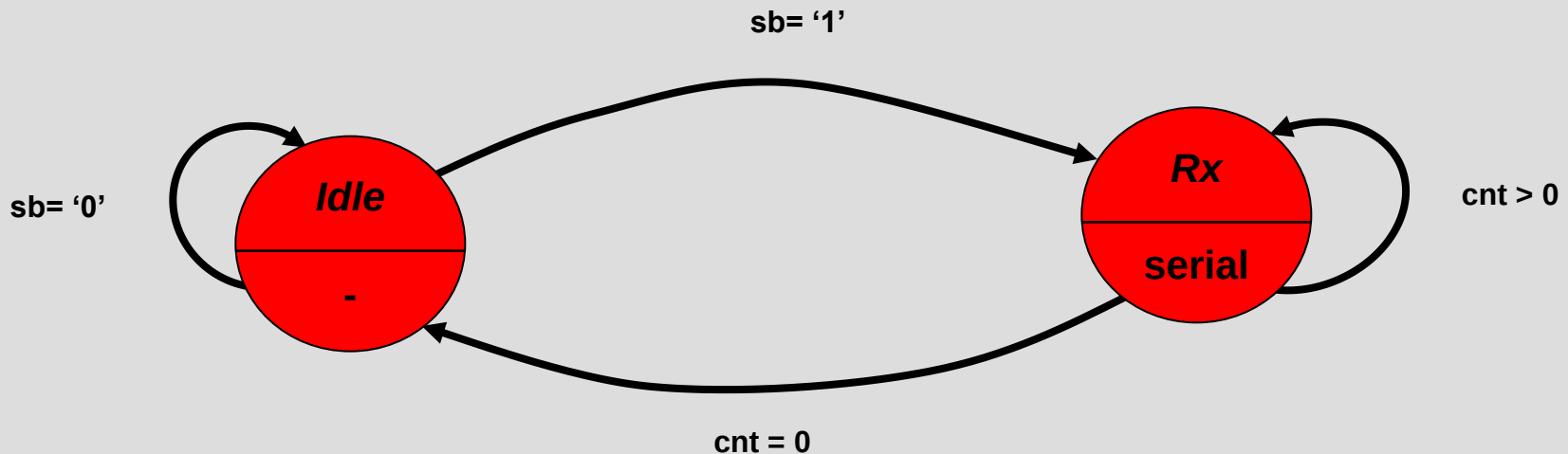


Implemente e simule um módulo *deserializer* em *VHDL*.

Implementações Síncronas

- Máquinas de Estado Finito (*FSM*):

- Exercício:



Testbenches Descritos em VHDL

Testbench

■ Motivação:

• Criar testes de projeto em *software*:

- Pode-se desenvolver códigos de alto nível (em *VHDL*) para se testar os módulos;
- Cria-se uma infinidade de possibilidades de depuração;
- Testes de transiente e de regime permanente;
- Testes de *DSP*, Computacionais, de interfaceamento, etc.

• Criar testes de projeto em *hardware*:

- Possibilidade de criação de gerador de sinais digitais para teste em *hardware*;
- Interfaceamento via *JTAG* ou pinos “normais”;
- Depuração do projeto em *hardware*.

Testbench

■ Sintaxe:

- A sintaxe de um *testbench* é idêntica a de uma integração, com a diferença de que existirá um processo gerando sinais de teste;

```
-- libraries
entity nome_da_entidade is
  port(
    -- declaração das portas da integração
  );
end entity nome_da_entidade;
```

Testbenches de *hardware* normalmente possuem portas para interface

```
-- libraries
entity nome_da_entidade is
end entity nome_da_entidade;
```

Testbenches de *software* não necessitam dessas portas, podendo-se então deixar a declaração da entidade sem a nomeação de nenhuma porta.

Testbench - Software

■ Gerenciamento de Sinais:

- Em um testbench é interessante passar para o código VHDL o máximo de funcionalidades possíveis;
- Quando utiliza-se o script TCL ou as interfaces gráficas de simuladores o processo de simulação torna-se mais lento pelo seu maior custo computacional;
- Além disso a possibilidade de tornar o teste autocontido (o próprio *testbench* gera os sinais e os avalia) diminui a influência humana padronizando os testes*.

* Facilitar a repetibilidade do teste é importante para submeter o módulos aos mesmos tipos de entradas e distúrbios, além de pode-se sempre aumentar o espaço de testes incrementando o *testbench*.

Testbench - Software

■ Gerenciamento de Sinais:

- Os principais sinais existentes em um *testbench* são o *clock* e o *reset*;
- Para implementar-se estes sinais existem uma técnica bastante simples para gerenciá-los;
- Utiliza-se constantes com o tipo *time* e o comando *after* (pode-se lançar mão dessas premissas, pois se está lidando com um código não prototipável);
- Basta então criar um sinal astável para o *clock* utilizando da mesma premissa utilizada em um oscilador;
- E criar um sinal monoestável para o *reset*, utilizando a idéia de um *timer*.

Testbench - Software

■ Gerenciamento de Sinais:

Gerenciamento do *Clock*

```
constant CLK_HALF_PER : time := 10 ns;  
signal clk : std_logic := '0';
```

Declaração feita normalmente (antes do *begin* da *architecture*)

Atribuição feita normalmente (na descrição comportamental do código)

```
clk <= not clk after CLK_HALF_PER;
```

Gerenciamento do *Reset*

```
constant RST_PER : time := 1 us;  
signal rst : std_logic := '1';
```

Declaração feita normalmente (antes do *begin* da *architecture*)

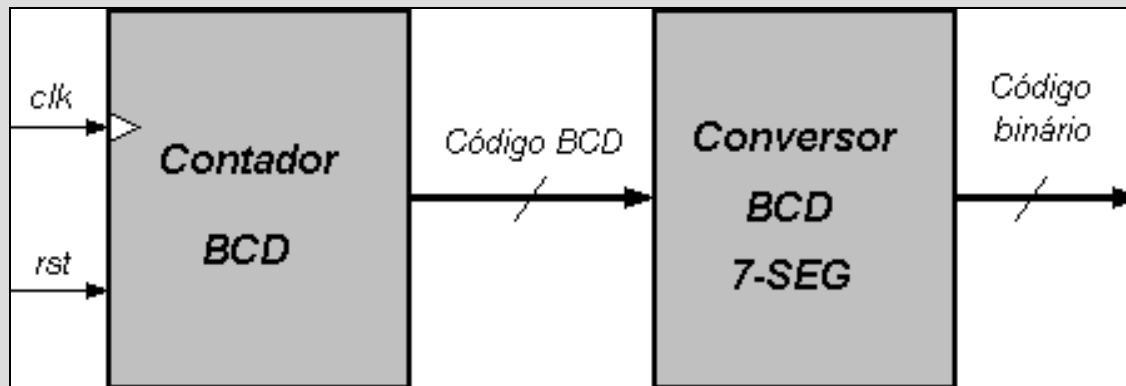
Atribuição feita normalmente (na descrição comportamental do código)

```
rst <= '0' after RST_PER;
```

Testbench - Software

■ Exercício:

Faça a integração dos módulos descritos a seguir

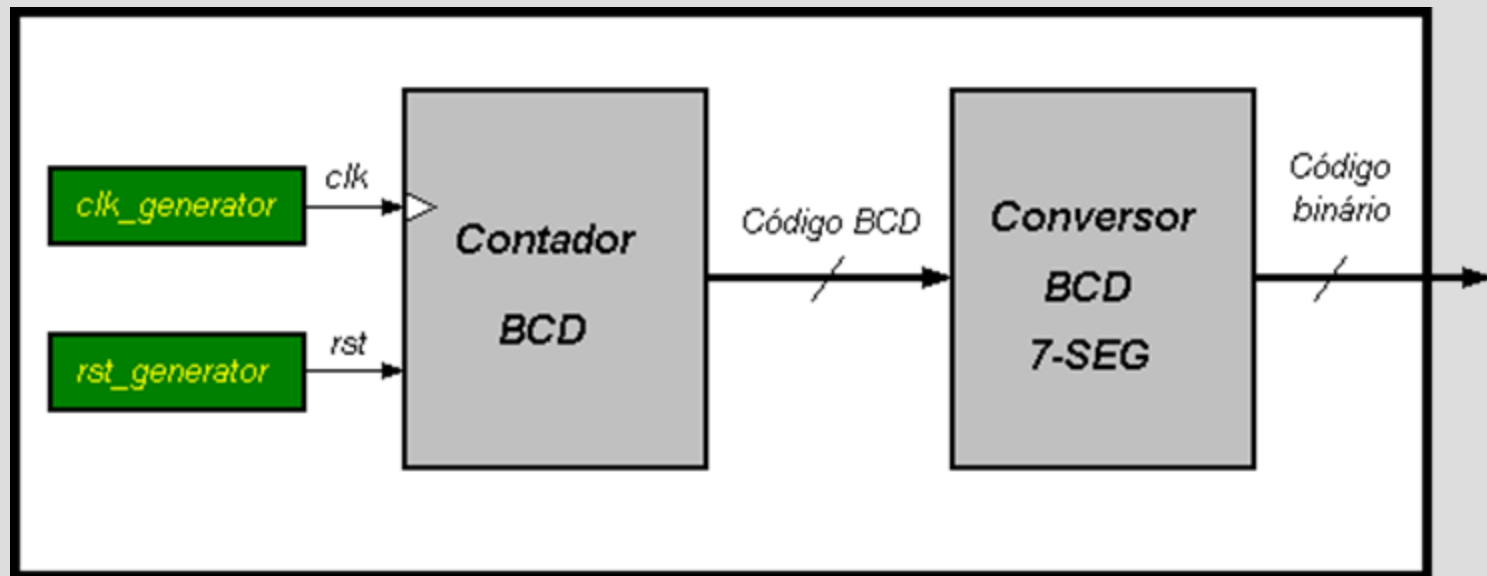


Códigos disponibilizados no arquivo ***conversor.zip***

Testbench - Software

■ Exercício:

Crie um *testbench* para o projeto anterior da seguinte maneira



LaPSE

Laboratório de Prototipação Digital
e Sistemas Embarcados

Universidade do Vale do Rio dos Sinos - UNISINOS

Prototipação em *PLDs*

Obrigado!