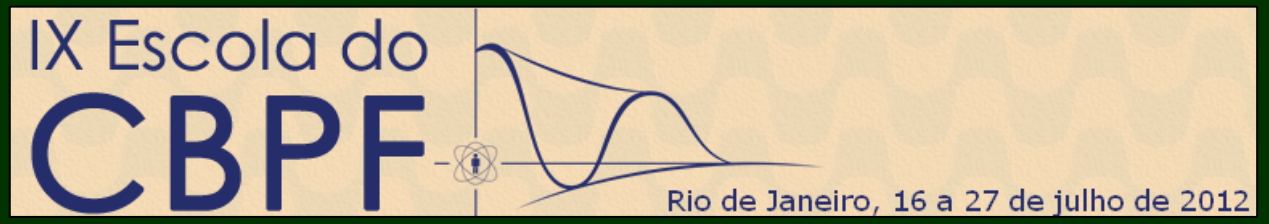


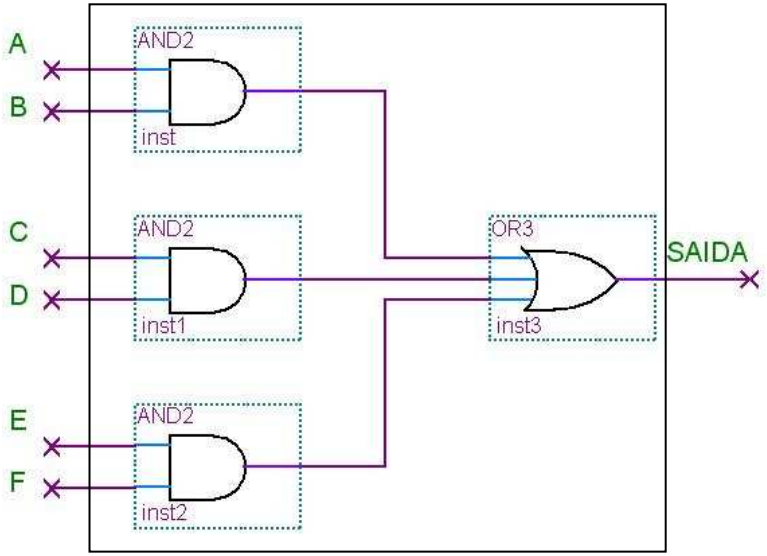

# G15 Eletrônica Digital para Instrumentação

Prof. Herman P. Lima Jr (hlima@cbpf.br)

Monitor: Rafael Gama

**Centro Brasileiro de Pesquisas Físicas**
**Ministério da Ciência, Tecnologia e Inovação (MCTI)**

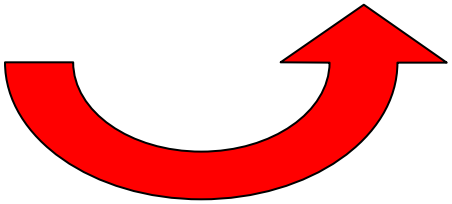
# Objetivo




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

entity logica is
   port (A,B,C : in  std_logic;
         D,E,F : in  std_logic;
         SAIDA : out std_logic);
end logica;

architecture v_1 of logica is
begin
   SAIDA <= (A and B) or (C and D) or (E and F);
end v_1;
```



# Referências


- **Fundamentals of Digital Logic with VHDL Design**, *Stephen Brown, Zvonko Vranesic,* McGraw-Hill, 2000.
- **The Designer's Guide to VHDL**, *Peter Ashenden*, 2nd Edition, Morgan Kaufmann, 2002.
- **VHDL Coding Styles and Methodologies**, *Ben Cohen*, 2nd Edition, Kluwer Academic Publishers, 1999.
- **Digital Systems Design with VHDL and Synthesis: An Integrated Approach**, *K. C. Chang*, Wiley-IEEE Computer Society Press, 1999.
- **Application-Specific Integrated Circuits**, *Michael Smith*, Addison-Wesley, 1997.

- **www.altera.com** (*datasheets, application notes, reference designs*)
- **www.xilinx.com** (*datasheets, application notes, reference designs*)
- **www.doulos.com/knowhow/vhdl_designers_guide** (*The Designer's Guide to VHDL*)
- **www.acc-eda.com/vhdlref/index.html** (*VHDL Language Guide*)
- **www.vhdl.org**

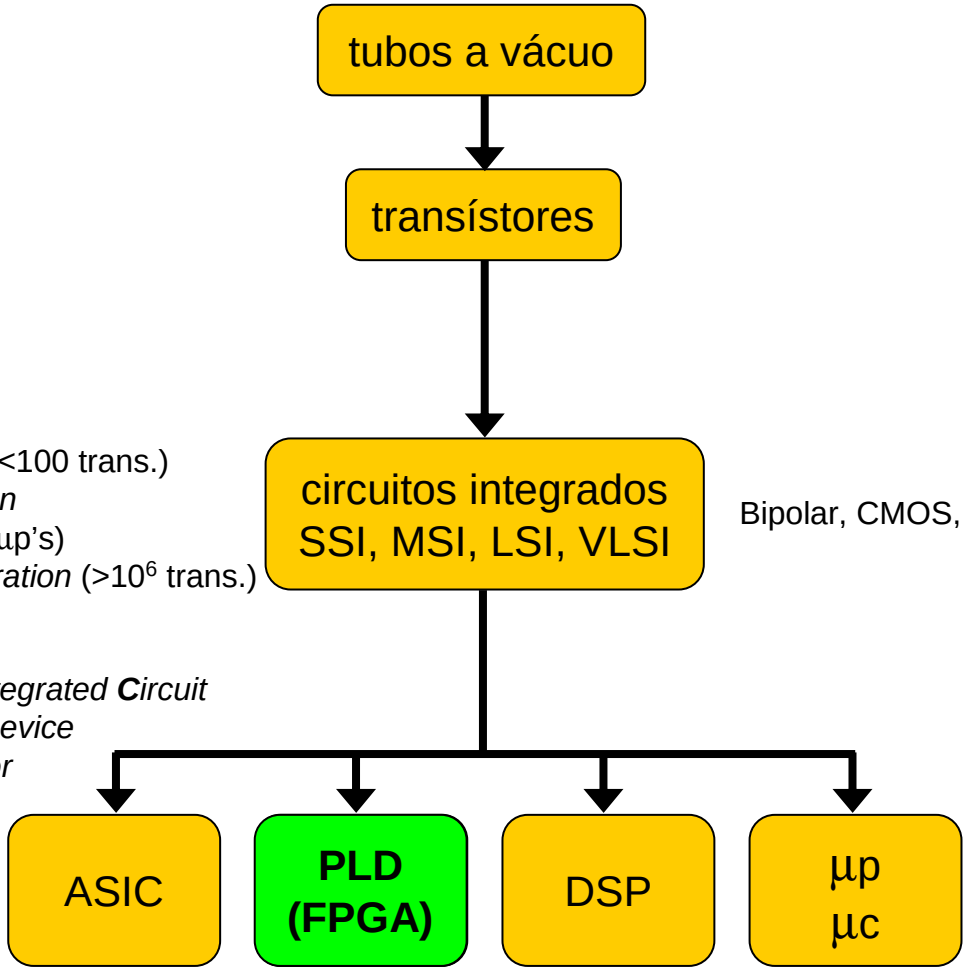
# Pré-requisito

- Eletrônica Digital:
  - ✓ portas lógicas
  - ✓ flip-flops
  - ✓ multiplexadores
  - ✓ comparadores
  - ✓ contadores

## Planejamento

- Eletrônica Digital – Histórico
- Eletrônica Digital – Tecnologias Atuais
- Lógica Programável
- VHDL – Introdução
- Aula Prática – Ferramenta EDA (*Electronic Design Automation*)

# Eletrônica Digital – Histórico




1945 → 1950 → 1958 → 1990 → hoje

Bipolar, CMOS, BiCMOS

SSI → *Small-Scale Integration* (<100 trans.)
MSI → *Medium-Scale Integration*
LSI → *Large-Scale Integration* (µp's)
VLSI → *Very Large-Scale Integration* (>$10^6$ trans.)

**ASIC** → ***A**pplication-**S**pecific **I**ntegrated **C**ircuit*
**PLD** → ***P**rogrammable **L**ogic **D**evice*
**DSP** → ***D**igital **S**ignal **P**rocessor*
**µp** → *Microprocessor*
**µc** → *Microcontroller*

# Eletrônica Digital – Tecnologias Atuais

| | ASIC | PLD | DSP | Micro contr. |
|---|---|---|---|---|
| Descrição, Aplicação | circuitos integrados dedicados (aplicação específica) | CPLD, FPGA (aplicação geral) | processadores matemáticos de alto desempenho | processadores, controladores |
| Reconfigurável ? | Não | Sim | Sim * (por instrução) | Sim * (por instrução) |
| Características básicas | Altíssima velocidade<br>Altíssima densidade<br>Baixo consumo | Alta velocidade<br>Alta densidade<br>Médio/alto consumo | Alta velocidade<br>Médio consumo | Média veloc.<br>Baixo consumo |
| Custo unitário | ↑ qtds ⇒ baixo<br>↓ qtds ⇒ elevado | ↑ qtds ⇒ alto<br>↓ qtds ⇒ baixo | ↑ qtds ⇒ médio<br>↓ qtds ⇒ médio | ↑ qtds ⇒ baixo<br>↓ qtds ⇒ baixo |
| Nível de descrição do projeto | Baixo / Médio (esquemático / textual) | Baixo / Médio (esquemático / textual) | Alto (textual) | Médio / Alto (textual) |

# Lógica Programável

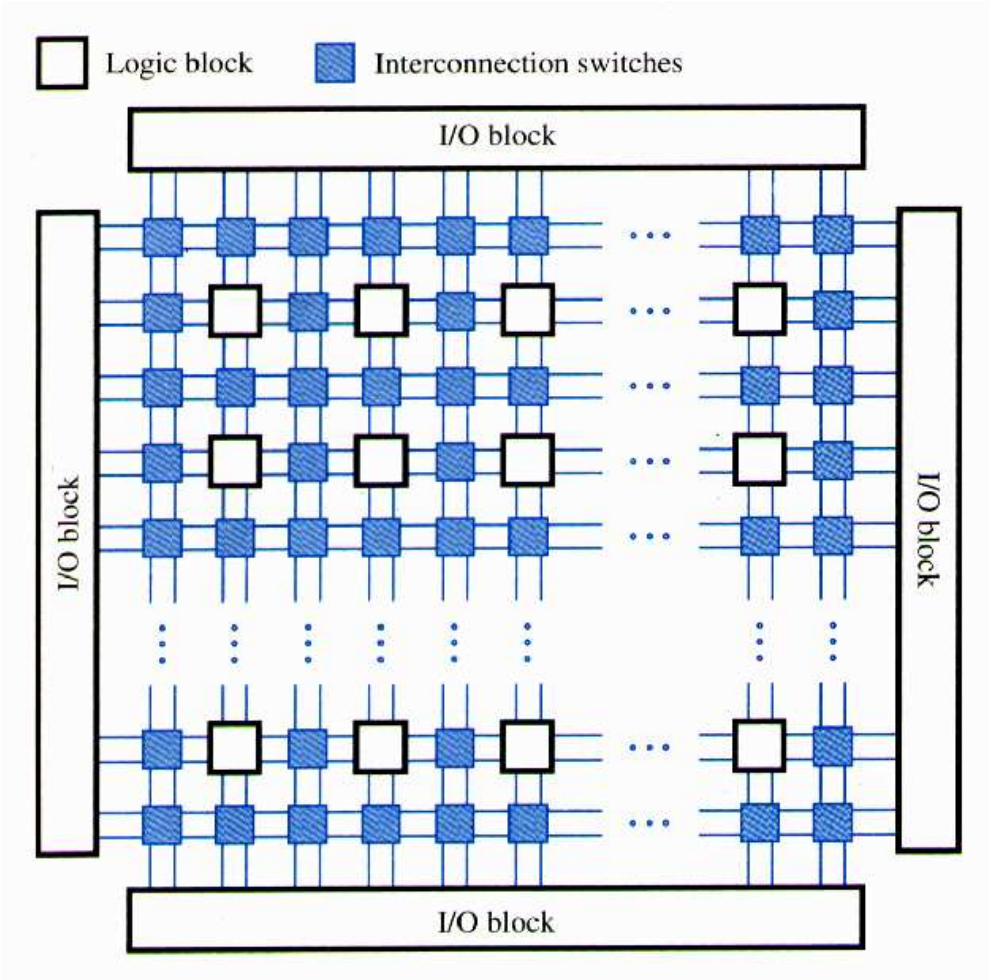
## Field Programmable Gate Array

- O que é $\Rightarrow$ dispositivo semicondutor lógico programável.
- Aplicações $\Rightarrow$ projetos de circuitos digitais de média ou grande complexidade (alta velocidade, alta densidade, segmentação de memória).
- Características
  - ✓ reprogramável, praticamente, por um número indefinido de vezes
  - ✓ tecnologias de configuração: SRAM, Antifuse ou Flash
  - ✓ utiliza memória externa (EEPROM) para configuração (tecnologia SRAM)
  - ✓ linguagens de descrição portáveis entre ferramentas (VHDL, Verilog)
  - ✓ alta densidade de lógica programável
  - ✓ alta densidade de registradores (ideal para circuitos síncronos)
  - ✓ biblioteca rica em blocos básicos (multiplexadores, decodificadores, ...)
  - ✓ blocos dedicados (DSP, memória, processadores, PLL, SERDES, ...)
  - ✓ diversos padrões digitais de comunicação (LVTTL, LVCMOS, LVDS, ...)
  - ✓ programável através de *IP Cores* (ex: interfaces de comunicação)
  - ✓ transição FPGA $\rightarrow$ ASIC pelo fabricante (ex: processo *Hardcopy* da Altera)

# Lógica Programável

## Estrutura da FPGA

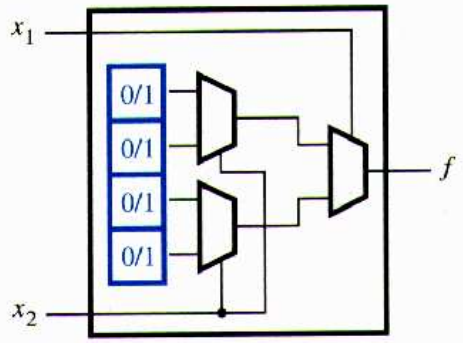
- Matriz bidimensional de blocos lógicos.
- Canais de conexão horizontal e vertical.
- Bloco lógico – contêm a lógica disponível.
- Bloco de I/O – comunicação com circuitos externos.
- Chaves de interconexão – conexão entre blocos lógicos, e entre blocos e pinos de I/O.



# Lógica Programável

## Blocos Lógicos

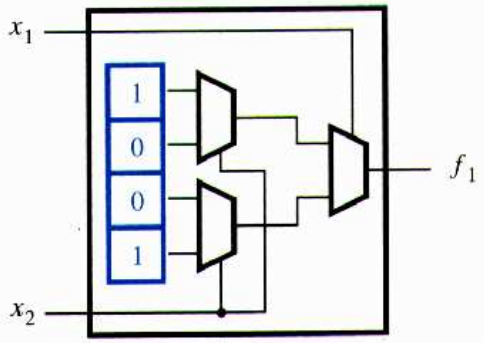
- *Look-Up Table* (LUT) – células de memória e multiplexadores.
- LUTs são utilizadas para implementar uma função lógica.
- Número de células de memória é igual a $2^{(\text{número de entradas})}$.
- Implementação transparente ao usuário.
- Células de memória volátil.




(a) Circuit for a two-input LUT

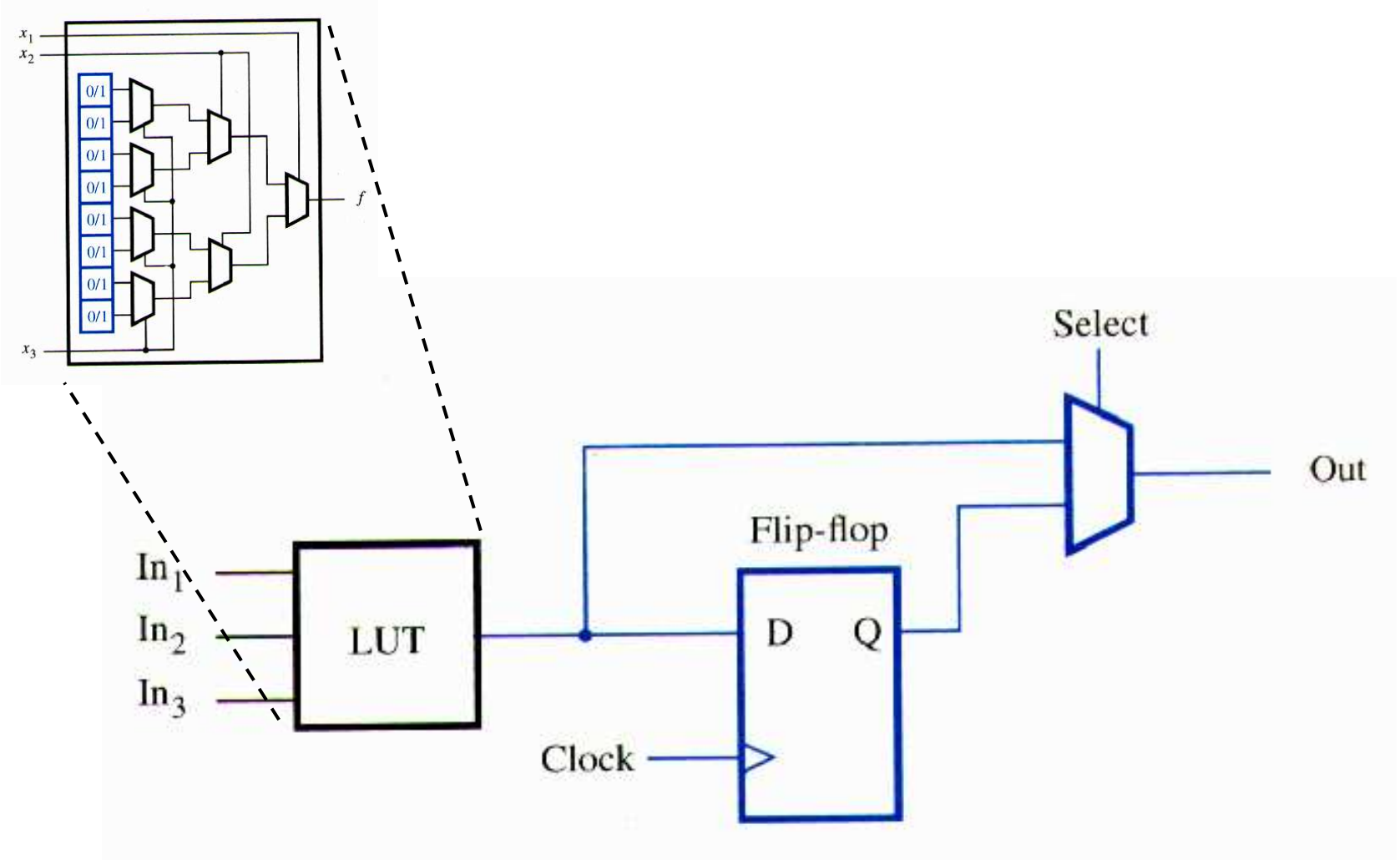
| $x_1$ | $x_2$ | $f_1$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

(b) $f_1 = \bar{x}_1\bar{x}_2 + x_1x_2$




(c) Storage cell contents in the LUT

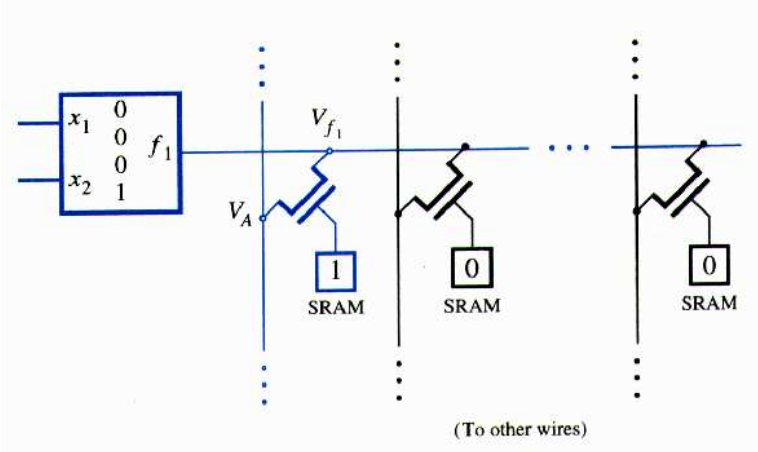
# Lógica Programável



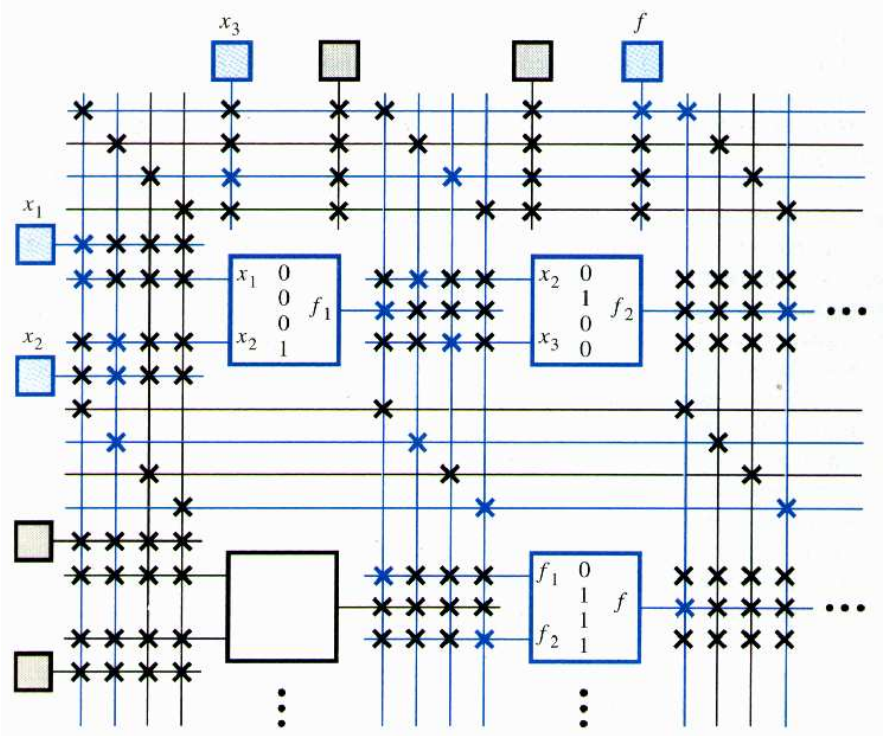

Exemplo de bloco lógico formado por uma LUT de 3 entradas e um registrador.

## Exemplo de projeto

- LUT de 2 entradas.
- 4 fios de interconexão.
- Células em azul ativadas.
- $f = f_1 + f_2 = x_1x_2 + x_2x_3$
- Chaves de interconexão são configuradas por células de memória SRAM.




Esquema de configuração dos fios de interconexão.




Exemplo de implementação da função combinacional.

# Lógica Programável

Exemplos atuais de FPGAs baixo custo e alta performance:

| | **Família Cyclone IV** (1) (16 opções) | **Família Stratix IV** (1) (17 opções) |
|---|---|---|
| **Tecnologia** | L=60 nm (1,2V ou 1,0V *core voltage*) | L=40 nm (0,9V *core voltage*) |
| **Elementos lógicos** | 6.272 - 149.760 | 72.600 - 813.050 |
| **Controle de *clock*** | 2 - 8 PLLs | 3 - 12 PLLs |
| **Blocos de memória** | 270 – 6.480 kbits | 6462 – 20.736 kbits |
| **Multiplicadores** | 15 - 360 (18 bits X 18 bits) | 384 - 1288 (18 bits X 18 bits) |
| **Transceivers** | 2 - 8 (3,125 Gbps) | 8 - 48 (11,3 Gbps) |
| **Pinos de I/O (máximo)** | 72 – 528 | 289 - 976 |
| **I/O programável** | sim | sim |
| **Preço unitário (US$)** | de US$12 a US$645 | US$800 a US$24.270 |

(1) *www.altera.com*

# VHDL - Introdução

- Breve histórico
- Padrões mais importantes para síntese
- Objetos em VHDL
- Descrição Estrutural
- Descrição Funcional
- Interface (*entity*)
- Implementação (*architecture*)

# VHDL - Introdução

- Pacotes
- Componentes
- Ações Simultâneas e Sequenciais
- Processos
- Palavras-chave
- Simulação

# VHDL - Introdução

- Modernos sistemas digitais podem ser complexos demais para descrição através de esquemáticos.
- No início da década de 80 surge a necessidade de outro método para descrever circuitos integrados muito complexos. O resultado é a criação das Linguagens Descritivas de Hardware (HDL's).
- Linguagens mais utilizadas: **VHDL** (Europa) e **Verilog** (EUA).
- Linguagens de mais alto nível de abstração já existem para modelar e verificar sistemas digitais complexos (ex: SystemVerilog e SystemC).

# VHDL - Introdução

| | VHDL 1993 | Verilog 1995 | SystemC | Verilog 2001 | System Verilog 3.1 | Verilog 2005 | VHDL 200X |
|---|---|---|---|---|---|---|---|
| Switch-level modeling | X | X | | X | X | X | X |
| ASIC timing | X | X | | X | X | X | X |
| Concurency | X | X | X | X | X | X | X |
| Design modularization | X | X | X | X | X | X | X |
| Gate-level modeling | X | X | X | X | X | X | X |
| Gate-level timing | X | X | X | X | X | X | X |
| Four-state logic | X | X | X | X | X | X | X |
| Event handling | X | X | X | X | X | X | X |
| Basic data types | X | X | X | X | X | X | X |
| Basic behavioral constructs | X | X | X | X | X | X | X |
| Dynamic generation of hardware | X | | | | X | X | X |
| Configurations | X | | | | X | | X |
| Simple assertions | X | | | | X | X | X |
| Assertions (formal methods) | | | | | X | X | X |
| Dynamic-memory allocation | X | | X | | X | X | X |
| Pointers | X | | X | | X | | X |
| Multidimensional arrays | X | | X | X | X | X | X |
| Records | X | | X | | X | X | X |
| Enumeration | X | | X | | X | | X |
| Automatic variables | X | | X | X | X | X | X |
| Signed numbers | X | | X | X | X | X | X |
| User-defined logic types | X | | | | | | X |
| User-defined resolution functions | X | | | | | | X |
| Void type | | | X | | X | | |
| Union | X | | X | | X | | X |
| Behavioral constructs | X | | X | | X | X | X |
| Classes with methods and inheritance | | | X | | X | | X |
| Sequential regular expressions | X | X | X | X | X | X | X |
| Temporal-property definitions | | | | | X | X | X |
| Scheduling for testbench and assertions | | | | | X | X | X |
| Semaphores | X | | | | X | | X |
| Stimulus generation | | | | | X | X | X |
| Constrained random data generator | | | | | X | X | |
| Coverage monitoring | | | | | X | X | X |
| Strings and strings operations | X | | | | X | | X |
| Standard C interface | | | | | X | X | X |
| Transaction modeling | | | X | | | X | X |
| Module encryption | | | | | | X | |

# VHDL - Introdução

- VHDL é uma linguagem para descrever sistemas eletrônicos digitais de média ou grande complexidade.
- A sigla significa: *VHSIC Hardware Description Language*, onde VHSIC significa *Very High Speed Integrated Circuit*.
- Criada a partir de um projeto norte-americano devido à necessidade de uma linguagem padrão para descrever a estrutura e a funcionalidade de circuitos integrados muito complexos.
- Adotada e padronizada pelo Instituto dos Engenheiros Elétricos e Eletrônicos (IEEE).
- A linguagem VHDL foi evoluindo ao longo dos anos e hoje o padrão mais utilizado é o IEEE Std.1076-1993, juntamente com o IEEE Std.1164-1993, que define um sistema de valores lógicos.

# VHDL - Introdução

- Características importantes:
  - Descrição estruturada, ou seja, um projeto é composto de sub-projetos e estes últimos são interconectados.
  - Especificação de funções utilizando formas similares de linguagens de programação.
  - Simulação de um projeto antes da fabricação de um circuito integrado (ASIC), ou da configuração de um dispositivo lógico programável (FPGA).

- Vantagens sobre esquemáticos:
  - Melhor legibilidade de um projeto. Possibilidade de particionar um projeto mais facilmente, desacoplando seus blocos.
  - Utilização de parâmetros que modificam capacidade e performance de um projeto, ou bloco.
  - Redução do custo de fabricação de protótipos. Redução do tempo de inserção de um novo produto no mercado.

- VHDL $\Rightarrow$ Modelagem - Simulação - **Síntese**

# VHDL - Introdução

## Padrões mais importantes para síntese com VHDL

- **IEEE 1076-1993**

  Define a base (núcleo) da linguagem para modelagem, simulação e síntese.

- **IEEE 1076.6-1999**

  Define o sub-conjunto destinado somente à síntese (*Register Transfer Level* - RTL).

- **IEEE 1164-1993** (STD_LOGIC)

  Define um padrão (*standard package*) de 9 valores lógicos para sinais:

  ‘U’ → *Unitialized*
  ‘X’ → *Forcing Unknown*
  ‘0’ → *Forcing 0*
  ‘1’ → *Forcing 1*
  ‘Z’ → *High Impedance*
  ‘W’ → *Weak Unknown*
  ‘L’ → *Weak 0*
  ‘H’ → *Weak 1*
  ‘-’ → *Don´t Care*

- **IEEE 1076.3** (*Numeric Standard*)

  Define, principalmente, os tipos de dados aritméticos **signed** e **unsigned**, junto com suas respectivas operações aritméticas, de deslocamento e de conversão.

# VHDL - Introdução

## Objetos em VHDL

- Existem 3 tipos de objetos: **sinais**, **constantes** e **variáveis**.
- O nome de um objeto pode utilizar qualquer caracter alfanumérico, desde que observadas as seguintes regras: (1) não pode ser uma palavra-chave de VHDL, (2) tem que iniciar com uma letra, (3) não pode terminar com *underscore* (_), e (4) não pode ter dois caracteres *underscore* juntos.
- Para síntese, os sinais (palavra-chave signal) são os mais importantes, pois representam os meios de comunicação entre blocos do projeto.
- Existem 3 locais onde um sinal pode ser declarado: na entidade, na parte de declarações de uma arquitetura, e na parte de declarações de um pacote.
- Declaração de um sinal: **signal** <nome_do_sinal> : [tipo] ;
- O tipo do sinal define os valores possíveis e sua utilização.

# VHDL - Introdução

## Tipos comuns dos objetos em VHDL

- **bit** e **bit_vector**
  - definidos nos padrões IEEE 1076 e IEEE 1164
  - o tipo **bit** pode assumir valores ‘0’ ou ‘1’
  - o tipo **bit_vector** é simplemente um *array* linear de objetos **bit**
  - ex: `signal c: bit_vector (1 to 4);` `c(1) <= ‘1’;` `c <= “1010”;`

- **std_logic** e **std_logic_vector**
  - definidos no padrão IEEE 1164
  - para utilizá-los, tem-se que incluir as seguintes linhas de código:

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

  - oferece maior flexibilidade que os tipos bit, podendo assumir valores ‘0’, ‘1’, ‘Z’, ‘-’, ‘L’, ‘H’, ‘U’, ‘X’ ou ‘W’
  - os valores ‘0’, ‘1’, ‘Z’ e ‘X’ são os mais úteis para síntese

# VHDL - Introdução

## Tipos comuns dos objetos em VHDL (cont.)

- **signed** e **unsigned**
  - definidos no padrão IEEE 1164, no pacote std_logic_arith;
  - este pacote também define a implementação dos operadores aritméticos (ex: +);
  - são similares ao tipo std_logic_vector, são *arrays* de std_logic;
  - têm como objetivo permitir a indicação no código de qual representação deve ser utilizada (sinalizada – complemento a 2 ou não-sinalizada);

- **integer**
  - definido para uso com operadores aritméticos (IEEE 1076);
  - o nº de bits não fica especificado no código, como um std_logic_vector;
  - por definição, um inteiro utiliza 32 bits, podendo assumir valores de $-(2^{31}-1)$ a $2^{31}-1$;
  - inteiros podem utilizar menos bits através da palavra-chave range:

```
signal x : integer range -127 to 127;
```

# VHDL - Introdução

## **Tipos comuns dos objetos em VHDL** (cont.)

- **boolean**
  - pode assumir os valores lógicos TRUE ou FALSE, equivalentes a ‘1’ e ‘0’;
  - ex: `signal flag : boolean ;`

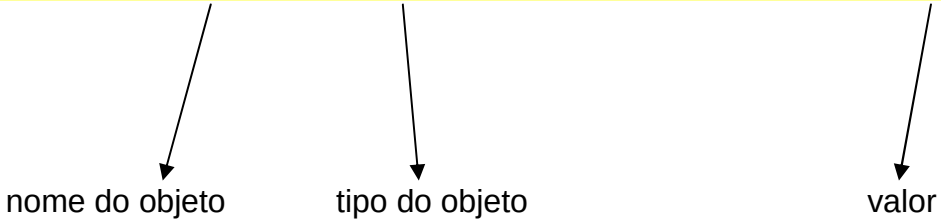
- tipo enumeração
  - tipo definido pelo projetista;
  - útil para definir estados no projeto de máquinas de estado;
  - ex:

```
type estados is (inicializa, processa);
signal y : estados ;
 ...
y <= processa;
```

## Constantes em VHDL

**constant**

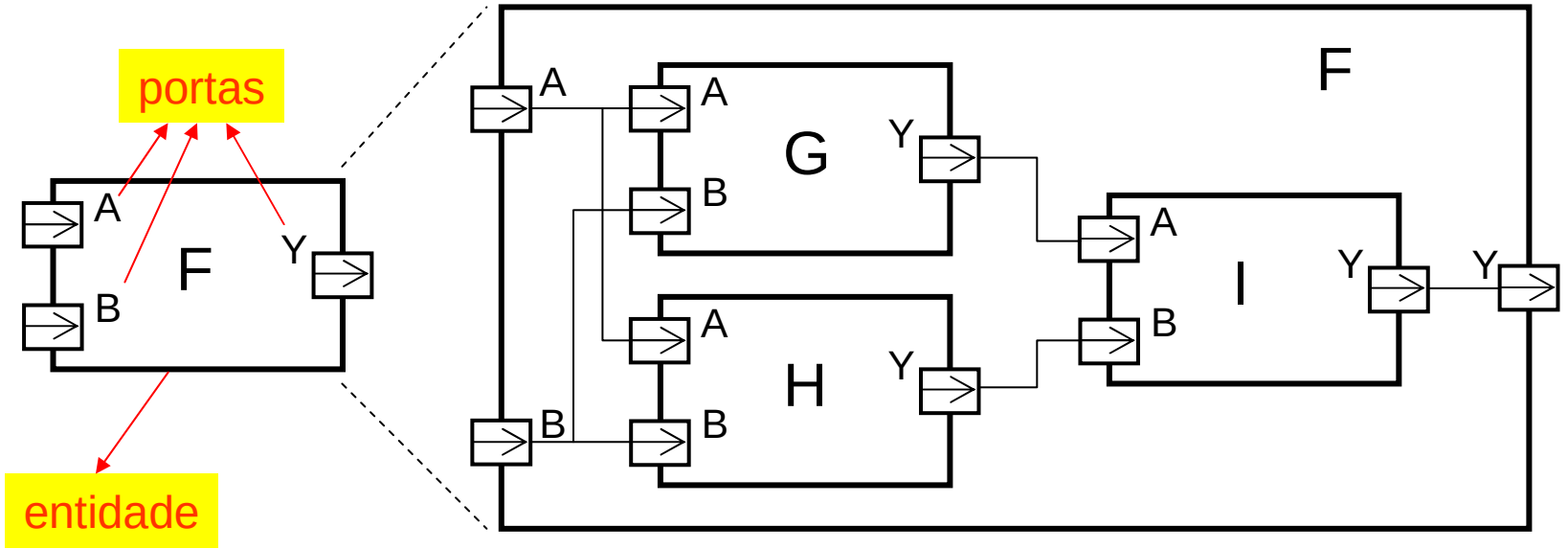
- um objeto tipo constant não pode mudar de valor ao longo do código;
- pode aparecer sem valor definido somente em pacotes (packages);
- útil para melhorar a legibilidade do código;
- ex: **constant** z : std_logic_vector(2 **downto** 0) := “011”;



# VHDL - Introdução

**Descrição ESTRUTURAL:**

- Um sistema eletrônico pode ser descrito como um módulo com entradas e saídas.
- Os valores nas saídas podem ser funções:
  - somente dos valores nas entradas em dado instante (circuito **combinacional**)
  - dos valores nas entradas e de estados internos (circuito **sequencial**)



# VHDL - Introdução

## Descrição FUNCIONAL:

- Um sistema eletrônico pode ser descrito simplesmente por sua função (ex: $Y = \overline{A} \cdot B + A \cdot \overline{B}$).
- Sistemas sequenciais obviamente não podem ser descritos unicamente como função de suas entradas.

Um projeto em VHDL pode ser baseado em componentes interconectados em hierarquia. Cada componente possui uma interface (entidade) e uma implementação (arquitetura), da seguinte forma:

componente (projeto)

interface

```
entity <nome_ent> is
    port (a,b: in bit;
          y: out bit);
end <nome_ent>;
```

implementação

```
architecture <nome_arq> of <nome_ent> is
begin
    ---
    ---
end <nome_arq>;
```

# VHDL - Introdução

## Interface (entity):

- Define as portas de acesso ao componente.
- Forma genérica de uma entidade:

```
entity <nome_entidade> is
    port ( <nome_sinal>: [modo] [tipo];
           <nome_sinal>: [modo] [tipo] );
end <nome_entidade>;
```

- Possíveis modos de um sinal:
  - **in** ⇒ o sinal é uma entrada para a entidade.
  - **out** ⇒ o sinal é uma saída para a entidade. O valor do sinal não pode ser usado dentro da entidade. Sua posição é sempre à esquerda do operador de atribuição <=.
  - **inout** ⇒ o sinal pode ser entrada ou saída para a entidade.
  - **buffer** ⇒ o sinal é uma saída para a entidade, mas seu valor pode ser lido dentro da entidade. Ele pode estar à esquerda ou à direita do operador de atribuição <=.

# VHDL - Introdução

## Implementação (architecture):

- Define a implementação de uma entidade.
- Forma genérica de uma arquitetura:

```
architecture <nome_arquitetura> of <nome_entidade> is
    [declaração de sinais]
    [declaração de constantes]
    [declaração de tipos]
    [declaração de componentes]
    [especificação de atributos]
begin
    [instância de componente]
    [ação simultânea]
    [processo]
    [geração]
end <nome_arquitetura>;
```

# VHDL - Introdução

Exemplo: comparador de 8 bits

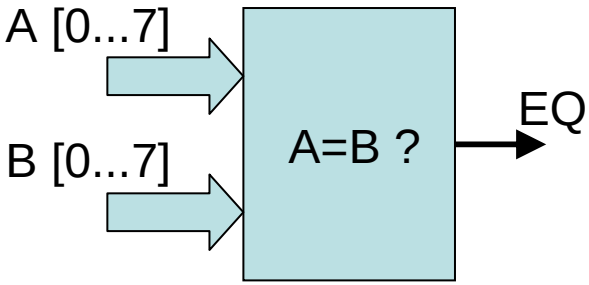
```
library ieee;

use ieee.std_logic_1164.all;

entity compare is
    port (A,B:   in std_logic_vector(0 to 7);
          EQ:   out std_logic);
end compare;

architecture one of compare is
begin
    EQ <= '1' when (A=B) else '0';
end one;
```

Representação esquemática:




- O circuito é **combinacional**.
- As palavras em destaque são palavras-chave em VHDL.
- VHDL não é *case-sensitive* (não diferencia letras maiúsculas de minúsculas).

# VHDL - Introdução

## PACOTES (package)

- **Entidades** e **Arquiteturas** são consideradas unidades básicas de um projeto.
- Unidades de projeto são segmentos de código que podem ser compilados separadamente e armazenados em uma biblioteca.
- Os 3 tipos de unidades mais úteis para síntese em VHDL são: entidades, arquiteturas e pacotes.
- Um pacote é designado pela palavra-chave package, e é utilizado para agrupar declarações de uso comum para diferentes unidades.
- Exemplo de um pacote:

```
package <nome_pacote> is

    constant NUM : integer := 16;

    type       MEM is array(0 to 31) of std_logic_vector(7 downto 0);

end <nome_pacote>;
```

## **PACOTES** - exemplo

- Pacote:

```
package cpu_types is
    constant word_size : positive := 16;
    constant address_size : positive := 24;
    subtype addr is bit_vector(address_size-1 downto 0);
end cpu_types;
```

- Uso do pacote:

```
use work.cpu_types.all;
entity address_decoder is
    port ( addr : in work.cpu_types.addr);
end address_decoder;
architecture functional of address_decoder is
    constant mem_low : work.cpu_types.addr := x"000000";
begin
    ---
end functional.
```

# VHDL - Introdução

## COMPONENTES (component)

- Componentes são considerados sub-circuitos de um projeto.
- Através de componentes é possível projetar um sistema em hierarquia.
- Para um bloco ser utilizado como componente de outro projeto, o componente tem que ser declarado e aplicado (criação de instâncias).
- Declaração de um componente:

```
component <nome_comp> is
port (a,b: in bit;
          y: out bit);
end component;
```

- Instância de um componente:

```
chip1: <nome_comp>
port map (a => data(1), b => data(0); y => out);
```

## COMPONENTES – exemplo

Comparador de 8 bits

```
library ieee;
use ieee.std_logic_1164.all;
entity compare is
      port (A,B:   in std_logic_vector(0 to 7);
                EQ: out std_logic);
end compare;
architecture one of compare is
begin
      EQ <= '1' when (A=B) else '0';
end one;
```

```
component <nome_comp> is
port (a,b: in bit;
           y: out bit);
end component;
```

```
chip1: <nome_comp>
port map (a => data(1), b => data(0); y => out);
```

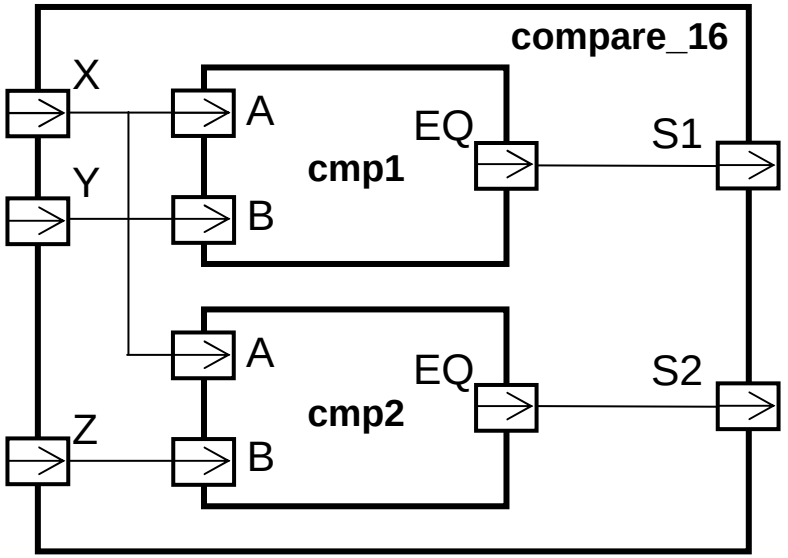
Comparador de 8 bits duplo

```
library ieee;
use ieee.std_logic_1164.all;
entity compare_16 is
      port (X,Y,Z:  in std_logic_vector(0 to 7);
               S1,S2: out std_logic);
end compare_16;
architecture arq of compare_16 is
   component compare is
   port (A,B: in std_logic_vector(0 to 7);
              EQ: out std_logic);
   end component;
begin
   cmp1: compare
   port map(A=>X, B=>Y, EQ=>S1);
   cmp2: compare
   port map(A=>X, B=>Z, EQ=>S2);
end arq;
```

## COMPONENTES – exemplo (cont.)

Comparador duplo de 8 bits

```
library ieee;
use ieee.std_logic_1164.all;
entity compare_16 is
      port (X,Y,Z:  in std_logic_vector(0 to 7);
            S1,S2: out std_logic);
end compare_16;
architecture arq of compare_16 is
  component compare is
  port (A,B: in std_logic_vector(0 to 7);
        EQ: out std_logic);
  end component;
begin
  cmp1: compare
  port map(A=>X, B=>Y, EQ=>S1);
  cmp2: compare
  port map(A=>X, B=>Z, EQ=>S2);
end arq;
```




- O circuito é **combinacional**.
- O circuito é **assíncrono**.
- As saídas S1 e S2 mudam de estado devido a mudanças nas entradas, no instante em que estas ocorrem (na prática existem atrasos de propagação dos sinais).

# VHDL - Introdução

## AÇÕES SIMULTÂNEAS E SEQUENCIAIS

- Um projeto em VHDL pode conter ações simultâneas e sequenciais (*concurrent assignments* e *sequential assignments*).
- Ações simultâneas são implementadas dentro da arquitetura.
- Ações sequenciais são implementadas dentro de processos.
- Existem 4 tipos de ações simultâneas em VHDL:
  - atribuição simples de sinal, atribuição selecionada de sinal, atribuição condicional de sinal e declaração geradora.
- **Atribuição simples de sinal:**

```
<nome_sinal>  <=  <expressão>;
```

  - O símbolo **<=** é chamado operador de atribuição em VHDL.
  - Exemplos:

```
f <= (x1 AND x2) OR x2;
f <= 'Z';
```

# VHDL - Introdução

## AÇÕES SIMULTÂNEAS (continuação)

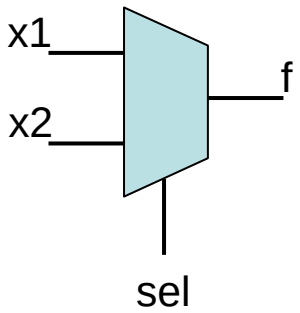
- **Atribuição selecionada de sinal:**

```
with <expressão> select
        <nome_sinal> <= <expressão> when <valor constante>{,
                        <= <expressão> when <valor constante>};
```

Exemplo:

```
signal x1,x2,sel,f : std_logic;
--
with sel select
        f <= x1 when '0',
             x2 when others;
```

→ multiplexador 2-1 →




- **Atribuição condicional de sinal:**

```
<nome_sinal> <= <expressão> when <valor constante> else
                {<expressão> when <valor constante>}
                <expressão>;
```

Ao contrário da atribuição selecionada, na atribuição condicional as condições não precisam ser mutuamente exclusivas, pois são testadas em ordem de prioridade.

## AÇÕES SIMULTÂNEAS (continuação)

- **Declaração geradora:** permite a repetição de uma função lógica ou de instância de componente. Pode ser usada da forma FOR ou IF.

```
for <variável índice> in <faixa> generate
        ação;
        {ação;}
end generate;
```

```
if <expressão> generate
        ação;
        {ação;}
end generate;
```

  – Exemplo:

```
for i in 0 to 3 generate
        bit: fulladd port map ( C(i), X(i), Y(i), S(i), C(i+1) );
end generate;
```

O código acima gera 4 instâncias de um componente chamado *fulladd*.

# VHDL - Introdução

## PROCESSOS

- Processos são utilizados para a implementação de **ações sequenciais**.
- Forma genérica:

```
{<nome_processo> :}
process (<nome_sinal>{,<nome_sinal>})
    [declaração de variáveis]
begin
    [estruturas WAIT]
    [atribuição de sinais]
    [atribuição de variáveis]
    [estruturas IF]
    [estruturas CASE]
    [estruturas LOOP]
end process [<nome_processo>];
```

lista de suscetibilidade:
⇒ as ações dentro do processo são executadas SOMENTE se um (ou mais) dos sinais da lista muda de estado.

- As ações são executadas na sequência em que ocorrem no código.
- Os sinais e variáveis mudarão de estado somente no final da execução do processo.

# VHDL - Introdução

## PROCESSOS (continuação)

Estrutura condicional IF:

```
if <expressão> then
    ação;
    {ação;}
elsif <expressão> then
    ação;
    {ação;}
else
    ação;
    {ação;}
end if;
```

Exemplo: multiplexador 2-1

```
if sel = '0' then
    f <= x1;
else
    f <= x2;
end if;
```

Estrutura CASE:

```
case <expressão> is
    when <valor constante> =>
    ação;
    {ação;}
    when <valor constante> =>
    ação;
    {ação;}
    when others =>
    ação;
    {ação;}
end case;
```

Exemplo: multiplexador 2-1

```
case sel is
    when '0' =>
        f <= x1;
    when '1' =>
        f <= x2;
end case;
```

## PROCESSOS (continuação)

Forma comum de um processo gerando ação síncrona:

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

entity top is
   port (clk, rst: in std_logic;
         data: in std_logic_vector(7 downto 0);
         q: out std_logic_vector(7 downto 0));
end top;

architecture top_arch of top is
begin
   reg:                                          <-- opcional
   process (rst,clk)
      variable qreg: std_logic_vector(7 downto 0);
   begin
      if rst = '1' then                          -- reset assíncrono
         qreg := "00000000";
      elsif (clk = '1' and clk'event) then
         qreg := data;                           -- ação síncrona
      end if;
      q <= qreg;
   end process;
end top_arch;
```

Representação esquemática:

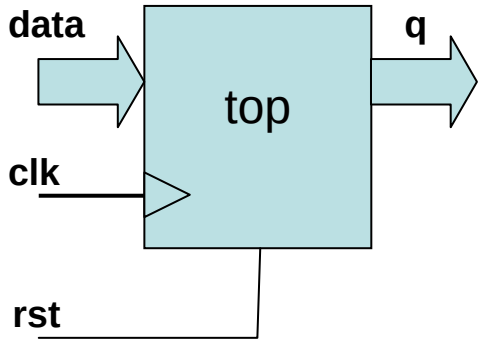
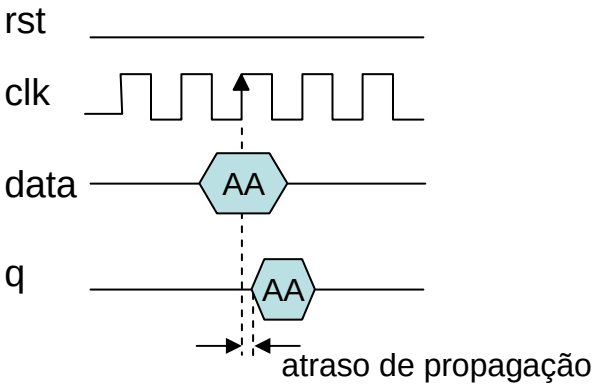



Diagrama temporal:



## Criando projetos reutilizáveis

- **Motivações:**
  - ✓ diminuir o tempo de elaboração de novos projetos
  - ✓ aumentar a produtividade
  - ✓ projetos mais eficientes (blocos repetidamente verificados)

- **Um projeto reutilizável tem que ser:**
  - ✓ pensado e projetado para resolver um problema genérico
  - ✓ bem codificado, comentado e documentado
  - ✓ exaustivamente verificado
  - ✓ independente da tecnologia (CPLD, FPGA)
  - ✓ independente da ferramenta de síntese
  - ✓ independente do simulador
  - ✓ independente da aplicação

## Criando projetos reutilizáveis

- **Recursos em VHDL que facilitam a reutilização:**

  - ✓ *Generics*
  - ✓ *Packages*
  - ✓ *Generate*
  - ✓ Objetos não-delimitados
  - ✓ Atributos

# VHDL - Introdução

## Generics

- Utilizados na criação de modelos (blocos) baseados em parâmetros.
- Os modelos possuem estrutura e funcionamento configuráveis.
- Exemplo: contador síncrono

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity counter is
   generic (  BIT_WIDTH              : integer    := 2;        - - estrutura
              COUNT_ENABLE           : boolean    := true;     - - estrutura
              DOWN_COUNT             : integer    := 0         - - funcionamento
            );
    port (  clk, rst, en  : in std_logic;
            count         : std_logic_vector(BIT_WIDTH-1 downto 0)   - - depende do parâmetro
         );
end counter;
```

# VHDL - Introdução

```
architecture one of counter is
    signal count_s   : std_logic_vector(BIT_WIDTH-1 downto 0);
    begin
      process(clk,rst)
      begin
        if (rst = '0') then
          count_s <= (others => '0');
        elsif (clk'event and clk='1') then
          if (COUNT_ENABLE = 1) then       - - sintetiza ou não o contador (estrutura)
            if (en = '1') then             - - habilita/desabilita a contagem (sinal externo)
              if (DOWN_COUNT = 0) then     - - contagem crescente ou decrescente (funcionamento)
                count_s <= count_s + 1;
              else
                count_s <= count_s – 1;
              end if;
            else
              count_s <= count_s;
            end if;
          end if;
        end if;
    end process;
    count <= count_s;
end one;
```

## Generics

Exemplo de uso do bloco *counter*: um contador crescente de 10 bits:

```
counter_10: counter
  generic map ( BIT_WIDTH          => 10;
                COUNT_ENABLE       => true;
                DOWN_COUNT         => false )
  port map ( clk => my_clk,
             rst => my_reset,
             en => my_en,
             count => count_out );
```

Observações:

- Se algum parâmetro não é declarado no *generic map*, assume seu valor padrão.
- O uso de parâmetros remove o uso de lógica não necessária durante a síntese.
- O uso de parâmetros é bem empregado em: tamanho de barramentos, registradores, capacidade de FIFOs, fator de divisão de *clocks* em divisores de frequência, etc.

## Pacotes

- São grupos de declarações que servem a um propósito comum.
- A visibilidade de um pacote é feita através da palavra-chave *use*.
- Vantagens em usar pacotes:
  - ✓ Se um novo parâmetro tem que ser incluído ou modificado, existe somente um arquivo (o pacote) que será modificado.
  - ✓ Melhor controle de um projeto.

Exemplo: criando um pacote com os parâmetros do contador *counter*

```
-- pacote de parâmetros par_pkg.vhd
package par_pkg is
        constant BIT_WIDTH              : integer := 10;
        constant COUNT_ENABLE           : boolean := true;
        constant DOWN_COUNT             : boolean := false;
end par_pkg;
```

## Pacotes

Exemplo: usando o pacote

```
-- uso do pacote de parâmetros par_pkg.vhd
library pkgs;
use pkgs.par_pkg.all;
entity counter is
port (  clk             : in std_logic;
        rst             : in std_logic;
        em              : in std_logic;
        count           : out std_logic_vector(BIT_WIDTH));
end counter;
architecture one of counter is
...
```

Observações:

- Um pacote de constantes tem o mesmo efeito do uso de parâmetros (*generics*) na modificação da estrutura e/ou do funcionamento durante a síntese.
- O uso de pacotes é recomendado somente para blocos que utilizam muitos parâmetros e que não serão replicados várias vezes em um mesmo projeto.

## Generate

- Permite a repetição de uma função lógica ou de componentes.
- Pode ser usada para criar, modificar ou remover estruturas condicionalmente.
- Exemplo de síntese condicional: saída síncrona ou assíncrona

```
constant SYNC_OUTPUTS : boolean := true;
```

```
sync: if (SYNC_OUTPUTS) generate
        process (clk)
          if (clk'event and clk = '1') then
            if (rd = '1') then
              q <= d;
            end if;
          end if;
        end process;
      end generate;
```

```
comb: if (not(SYNC_OUTPUTS)) generate
        process (rd)
          if (rd = '1') then
            q <= d;
          end if;
        end process;
      end generate;
```

- Em resumo, *generate* é um poderoso recurso para incluir ou excluir lógica em um projeto, tornando-o mais reutilizável.

## Objetos não-delimitados

- Exemplo: contador utilizando um barramento de saída não delimitado

```
entity counter is
   port (  clk          : in std_logic;
           rst          : in std_logic;
           count        : std_logic_vector );                    - - barramento não-delimitado
end counter;
architecture one of counter is
  signal        count_s : std_logic_vector(count'range)   - - mesma largura de 'count'
begin
  process (clk,rst)
  begin
     if (rst = '0') then
        count_s <= (count'range => '0');
     elsif (clk'event and clk = '1') then
        count_s <= count_s + 1;
     end if;
    ...
```

## Objetos não-delimitados

- Notar que o contador anterior não é sintetizável por si só. Ele TEM que estar incluído como um componente (sub-circuito) em outro projeto, como abaixo:

```
...
architecture inst of top is
  signal   my_count   : std_logic_vector(7 downto 0)   - - irá definir a capacidade do contador
  signal   clk, rst       : std_logic;
  cnt : counter
    port map ( clk => clk,
               rst => rst,
               count => my_count );              - - define automaticamente na síntese a
...                                              - - capacidade do contador
```

## Atributos

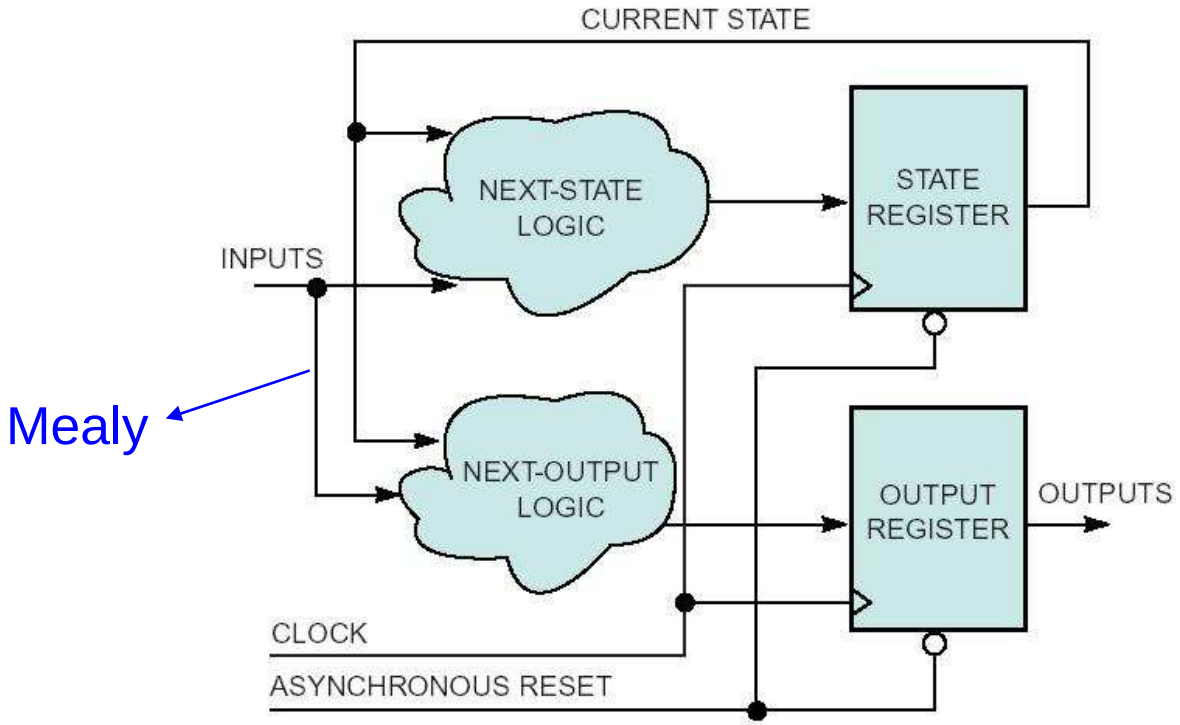
- Alguns atributos em VHDL são úteis na criação de projetos reutilizáveis, como:
  - ‘left - primeiro valor de um objeto
  - ‘right - último valor
  - ‘range - faixa de valores possíveis de um tipo ou objeto
  - ‘length - comprimento
  - ‘low - menor valor possível
  - ‘high - maior valor possível
- Exemplo de atributos para um tipo escalar:

```
type resistance is range 0 to 1E9

resistance’left = 0
resistance’right=1E9
resistance’low=0
resistance’high=1E9
```

## Máquinas de Estado (FSM)

- Muito utilizadas em projetos de circuitos síncronos.
- Utilize máquinas de estado com saídas e estados registrados.
- Esta abordagem tem as seguintes vantagens:



## Máquinas de Estado (FSM)

Exemplo:

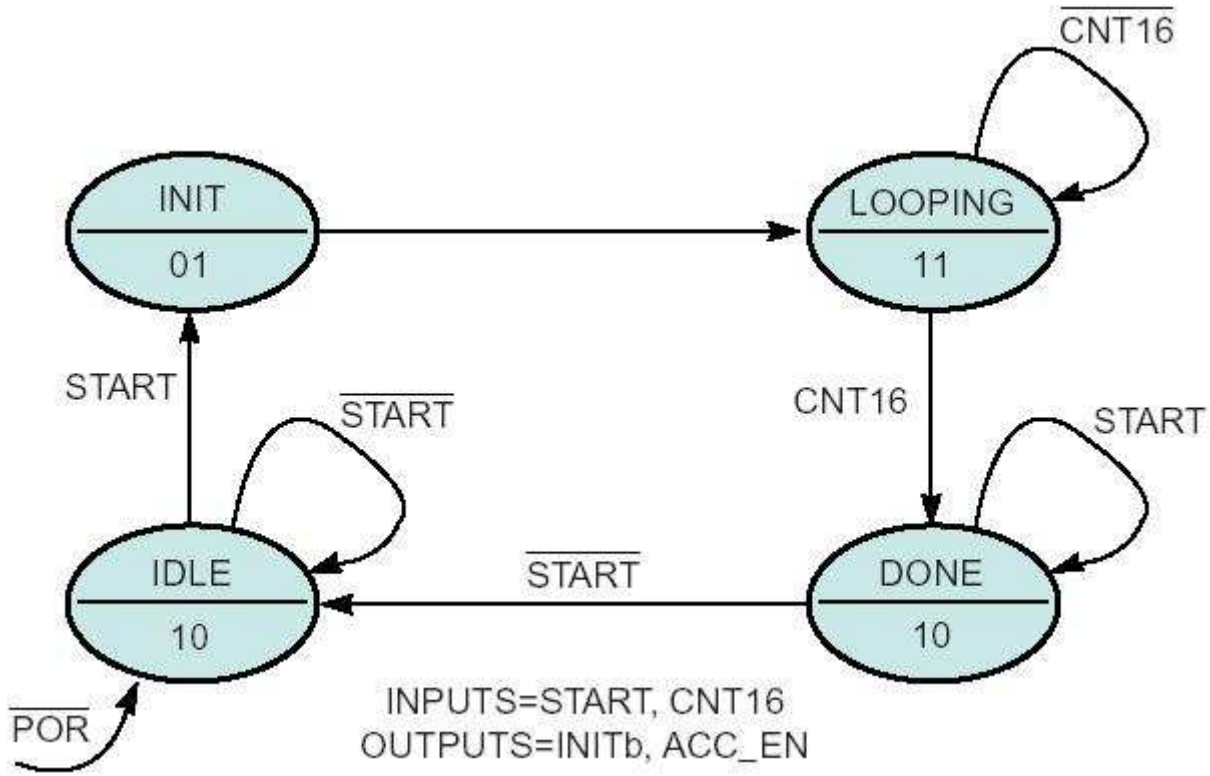



*Diagrama de estados*

# VHDL - Introdução

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3
4  entity fsm is
5    port(
6      CLK:    in  STD_LOGIC;
7      MRb:    in  std_logic;
8      START:  in  std_logic;
9      CNT16:  in  std_logic;
10     INITb:  out std_logic;
11     ACC_EN: out std_logic);
12 end fsm;
13
14 architecture RTL of fsm is
15
16   type STATE_TYPE is (IDLE,INIT,LOOPING,DONE);
17   signal current_state: STATE_TYPE;
18   signal next_state:    STATE_TYPE;
19   signal iINITb:        STD_LOGIC;
20   signal iACC_EN:       STD_LOGIC;
21
22 begin -- architecture
```

```
23
24   next_state_comb: process(current_state,START,CNT16)
25   begin
26     case current_state is
27       when IDLE =>
28         if START = '1' then
29           next_state <= INIT;
30         else
31           next_state <= IDLE;
32         end if;
33       when INIT =>
34         next_state <= LOOPING;
35       when LOOPING =>
36         if CNT16  = '1' then
37           next_state <= DONE;
38         else
39           next_state <= LOOPING;
40         end if;
41       when DONE =>
42         if START = '0' then
43           next_state <= IDLE;
44         else
45           next_state <= DONE;
46         end if;
48     end case;
49   end process;
```

*Código em VHDL de uma FSM* (1/2)

# VHDL - Introdução

```
50
51  output_comb: process(next_state)
52  begin
53    case next_state is
54      when IDLE =>
55        iINITb <= '1';
56        iACC_EN <= '0';
57      when INIT =>
58        iINITb <= '0';
59        iACC_EN <= '1';
60      when LOOPING =>
61        iINITb <= '1';
62        iACC_EN <= '1';
63      when DONE =>
64        iINITb <= '1';
65        iACC_EN <= '0';
66    end case;
67  end process;
68
69  state_flops: process(CLK,MRb)
70  begin
71    if (MRb = '0') then
72      current_state <= IDLE;
73    elsif (CLK'event and CLK = '1') then
74      current_state <= next_state;
75    end if;
76  end process;
```

```
77
78  output_flops: process(CLK,MRb)
79  begin
80    if (MRb = '0') then
81      INITb <= '0';
82      ACC_EN <= '0';
83    elsif (CLK'event and CLK = '1') then
84      INITb <= iINITb;
85      ACC_EN <= iACC_EN;
86    end if;
87  end process;
88
89  end RTL; -- architecture
```

*Código em VHDL de uma FSM* (2/2)

# VHDL - Introdução

**PALAVRAS-CHAVE** (IEEE Std.1076-1993)

abs
access
after
alias
all
and
architecture
array
assert
attribute

begin
block
body
buffer
bus

case
component
configuration
constant

disconnect
downto

else
elsif
end
entity
exit

file
for
function

generate
generic
group
guarded

if
impure
in
inertial
inout
is

label
library
linkage
literal
loop

map
mod

nand
new
next
nor
not
null

of
on
open
or
others
out

package
port
postponed
procedure
process
protected
pure

range
record
register
reject
rem
report
return
rol
ror

select
severity
shared
signal

sla
sll
sra
srl
subtype

then
to
transport
type

unaffected
units
until
use

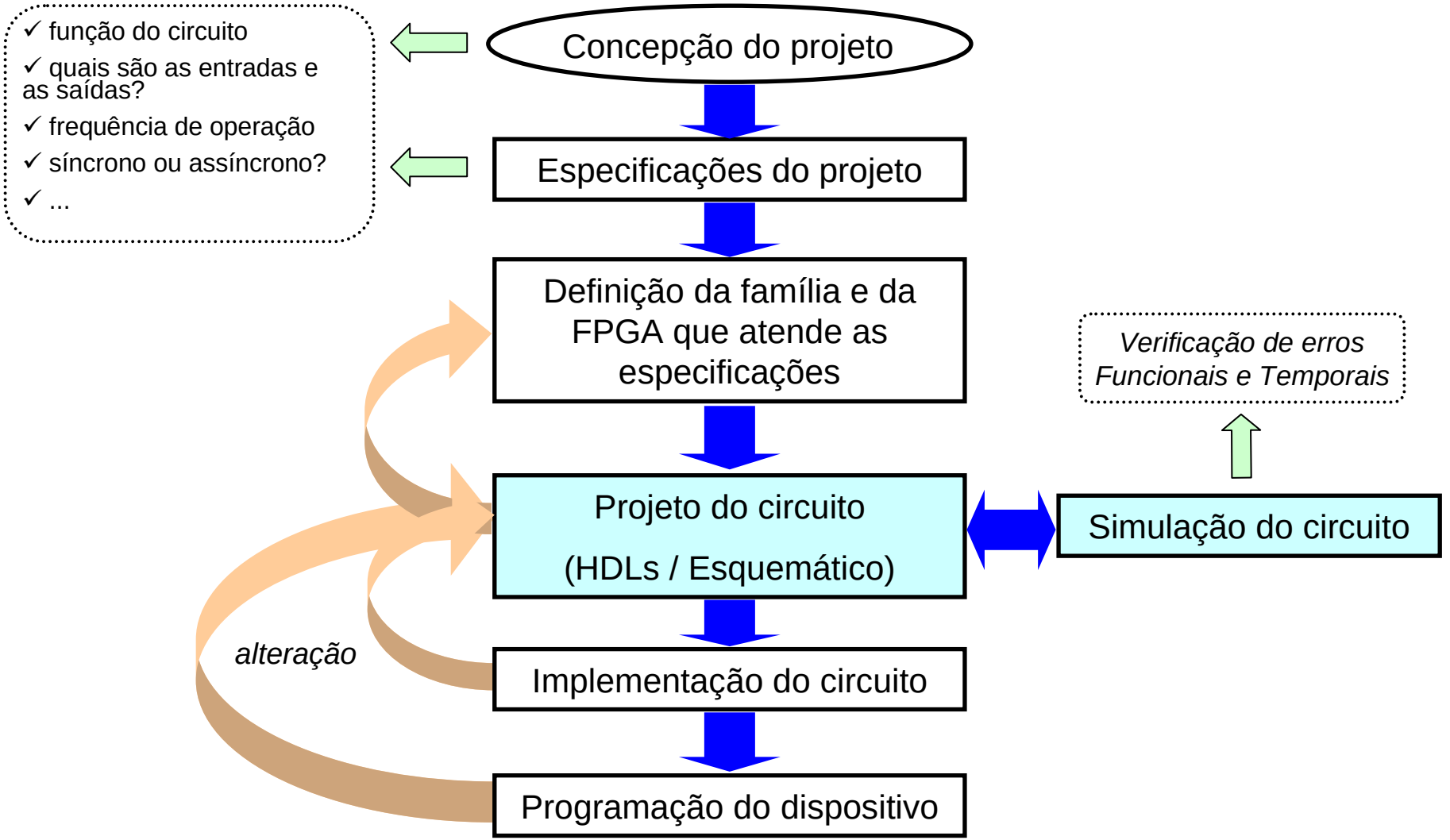
variable

wait
when
while
with

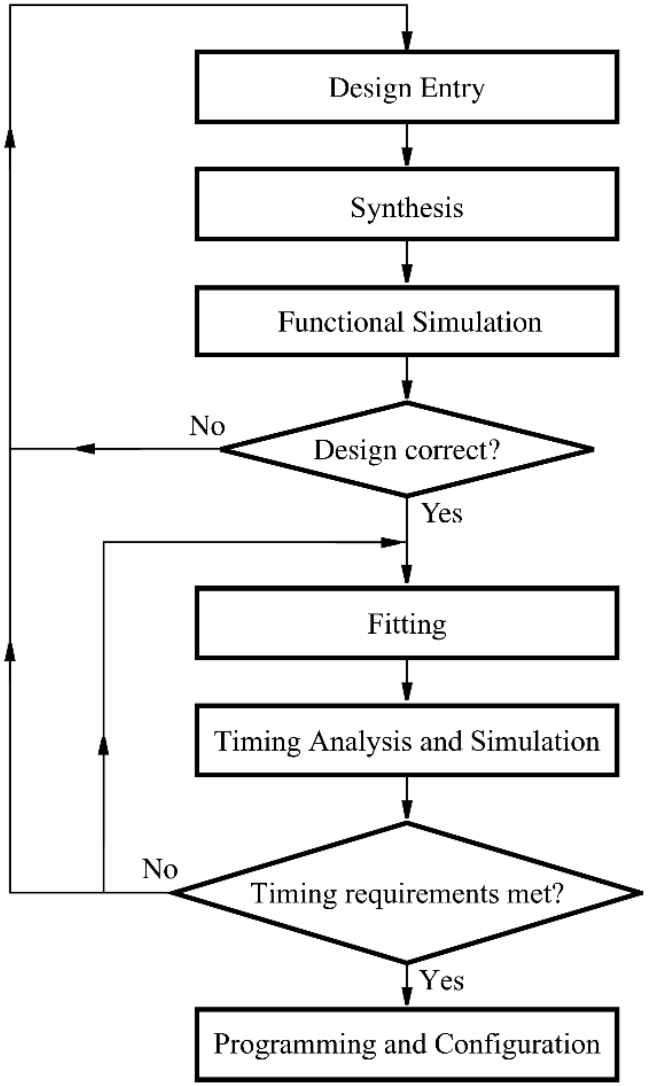
xnor
xor

# Projetos com Lógica Programável

## Fluxo de um projeto com lógica programável



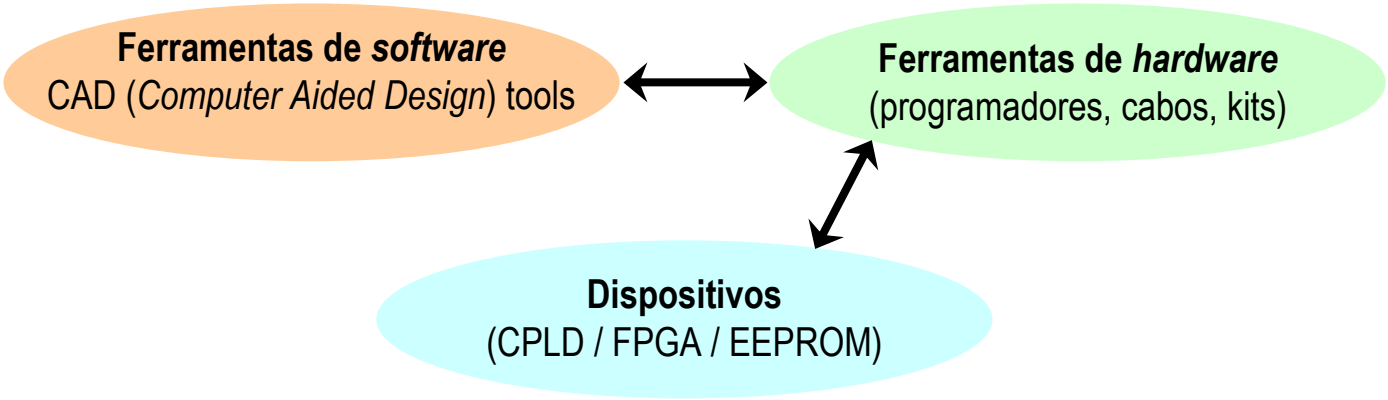
## Fluxo de CAD (*Computer Aided Design*)




- **Design Entry**
  o circuito desejado é especificado (descrito) através de uma HDL (Verilog ou VHDL) ou esquemáticos
- **Synthesis**
  o circuito é sintetizado pela ferramenta em uma *netlist* que define os elementos lógicos (LEs) necessários para realizar o circuito e as conexões entre eles
- **Funcional Simulation**
  o circuito sintetizado é testado para verificar seu funcionamento lógico (não são considerados atrasos)
- **Fitting**
  o posicionamento dos LEs dentro da FPGA é definido. As conexões de roteamento também são definidas pela ferramenta.
- **Timing Analysis**
  os atrasos de propagação no circuito do *fitting* são analisados, fornecendo resultados de desempenho do projeto.
- **Timing Simulation**
  o circuito do *fitting* é testado de forma funcional e temporal

# Projetos com Lógica Programável

## Elementos necessários para um projeto com lógica programável




- **Ferramentas CAD:** utilizadas para a projeto, simulação e implementação do sistema. Exemplos: Quartus II (Altera) e ISE (Xilinx).
- **Ferramentas de *hardware*:** interface entre a ferramenta CAD e o dispositivo.
- **Dispositivos:** os circuitos integrados que implementam o projeto.

## Simulação

- **Test bench:**
  - É um projeto a parte utilizado somente para simular um projeto.
  - É formado por uma entidade e uma arquitetura.
  - A entidade de um Test Bench não possui portas.
  - Geralmente contém 2 processos: um processo gerador de *clock* e um processo que gera os outros estímulos.

- **Dicas de simulação:**
  - Não esqueça NENHUMA entrada do circuito simulado sem estímulos.
  - Simule do circuito menos complexo para o mais complexo, bloco a bloco.
  - Após cada modificação no projeto, uma nova simulação tem ser feita.
  - Salve os resultados de simulação quando julgar importante.

## Simulação

Test Bench para o registrador de 8 bits.

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

entity test is
end test;

architecture one of test is
  component top is
    port (clk, rst: in std_logic;
          data: in std_logic_vector(7 downto 0);
          q: out std_logic_vector(7 downto 0));
  end component;
  signal clk, rst: std_logic;
  signal data, q: std_logic_vector(7 downto 0);
begin
  DUT: top port map (clk, rst, data, q);
```

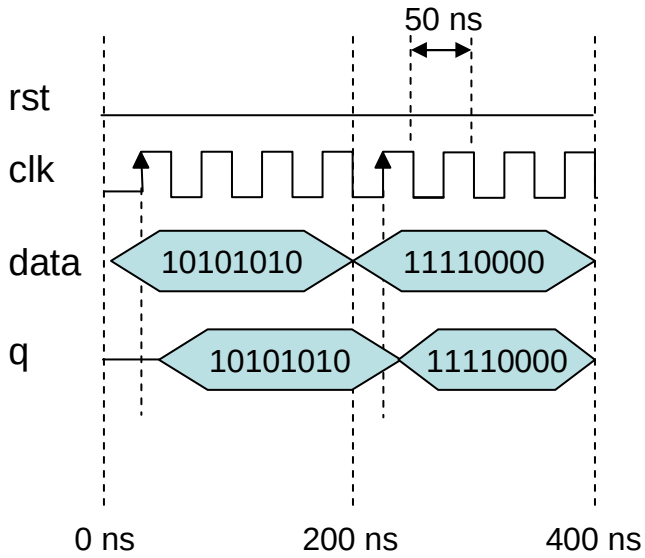
```
  clock: process
    variable clktmp: std_logic := '1';
  begin
    clktmp := not(clktmp);
    clk <= clktmp;
    wait for 25 ns;
  end process;
  stimulus: process
  begin
    rst <= '0';
    data <= "10101010";
    wait for 100 ns;
    data <= "01010101";
    wait for 200 ns;
  end process;
end one;
```

## Simulação

Exemplo de Test Bench: testando o registrador de 8 bits.

```
  clock: process
    variable clktmp: std_logic := '1';
  begin
    clktmp := not(clktmp);
    clk <= clktmp;
    wait for 25 ns;
  end process;
  stimulus: process
  begin
    rst <= '0';
    data <= "10101010";
    wait for 200 ns;
    data <= "11110000";
    wait for 200 ns;
  end process;
end behavior;
```

Resultado da simulação:



# boa sorte