Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática - FACIN

LABORG

Prof. Dr. Rafael Garibotti

❖ Baseado no material cedido pelos Profs. **Dr. Fernando Moraes** e **Dr. Ney Calazans**

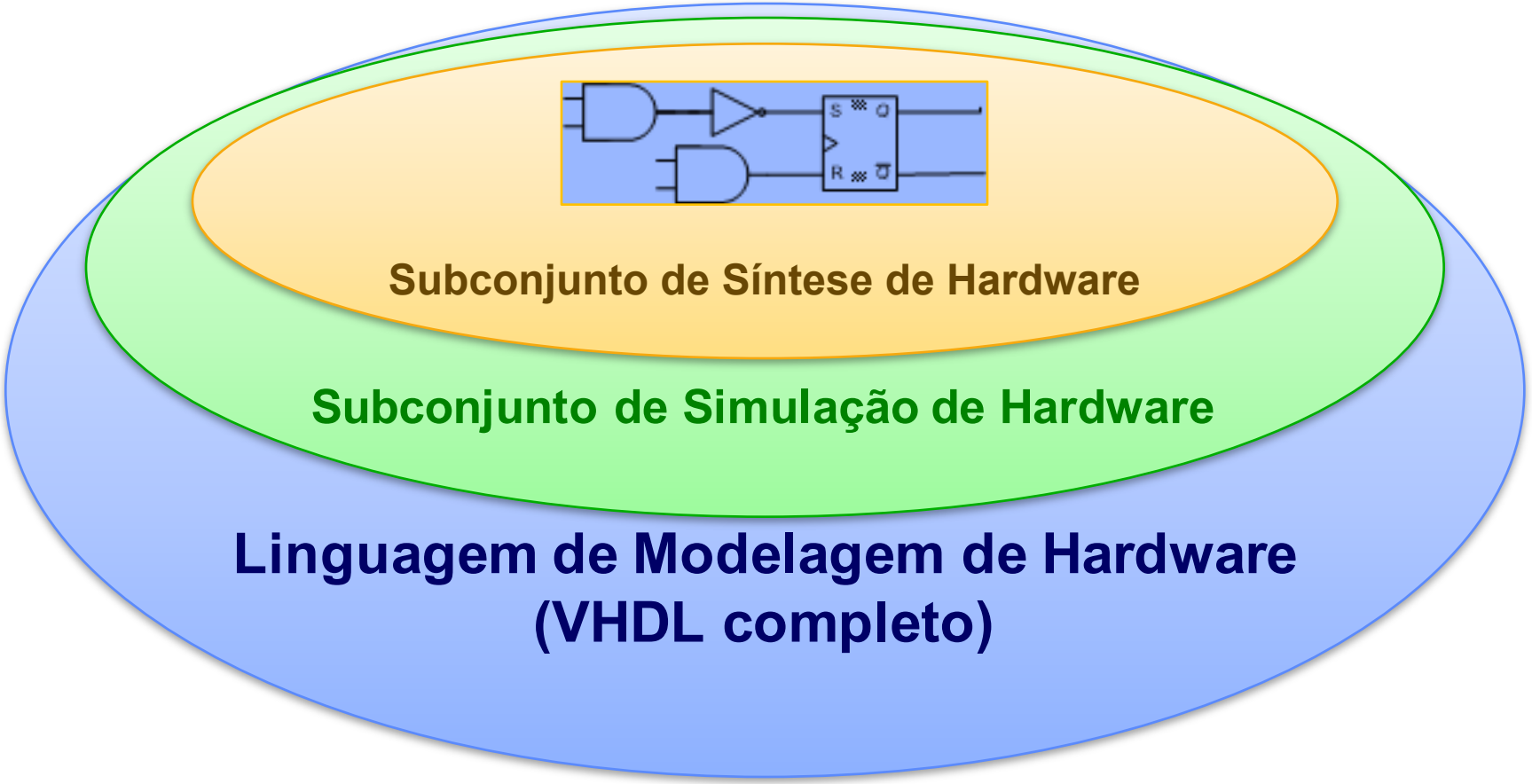
AULA SOBRE:

PROCESSOS, PARALELISMO E COMANDO PROCESS

INTRODUÇÃO

INTRODUÇÃO

- VHDL pode ser visto como formada por 3 linguagens, o todo e dois subconjuntos próprios.



INTRODUÇÃO

- Linguagem de Modelagem de Hardware
  - ✓ Aceita todas as construções da linguagem, incluindo aquelas não simuláveis temporalmente e as não sintetizáveis.
  - ✓ Por exemplo, o que acontece ao tentar simular o texto ao lado?

1. Este texto compila no simulador, mas é impossível de ser simulado usando formas de onda. Por quê?
2. Porque esse é um código atemporal, não temos um **wait**, ou um tempo de referência no process!!!

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity nao_sim is
end nao_sim;

architecture nao_sim of nao_sim is
  signal y : integer := 0;
begin
  process
    variable x : integer :=1;
  begin
    if x<10 then
      x := x+1;
    end if;
    y <= x;
  end process;
end nao_sim;
```

INTRODUÇÃO

- Linguagem de Simulação
  - ✓ Aceita parte das construções da linguagem de modelagem, incluindo descrições de hardware sintetizáveis e os testbenches usados para validá-lo.
  - ✓ Por exemplo, o que acontece ao tentarmos sintetizar o código abaixo no ISE?

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity nao_sint is
  port ( entra: in std_logic;
         sai: out std_logic);
end nao_sint;
architecture nao_sint of nao_sint is
begin
  sai <= not entra after 10ns;
end nao_sint;
```

1. Este texto compila OK no simulador, simula OK, mas é impossível de sintetizá-lo desta forma e ele operar como previsto. Por quê?
2. Porque ele não sabe qual lógica combinacional colocar nestes primeiros 10ns.

PROCESSOS EM VHDL

PROCESSOS EM VHDL

- O que é um processo em VHDL?
  - ✓ Dito de forma muito simples e genérica, um processo é um "pedaço de hardware".
- Como cada pedaço de hardware opera em paralelo com outros pedaços de hardware existentes, a noção de paralelismo é fundamental em VHDL.
- Por outro lado, a conexão entre dois pedaços de hardware cria "comunicação" entre estes.
  - ✓ Observação: o sequenciamento de eventos de um pedaço de hardware que gera sinais que são entrada de um outro pedaço de hardware afeta o sequenciamento de eventos do último.

PROCESSOS EM VHDL

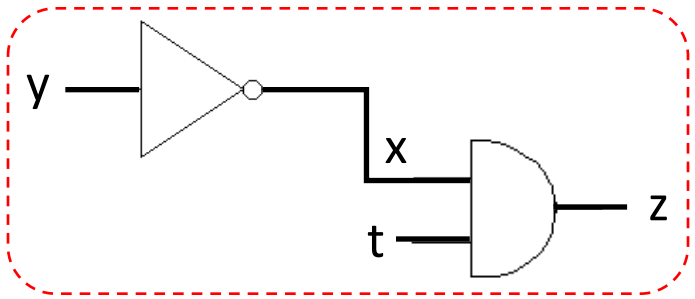
- Não confundir a noção geral de processo VHDL com o comando ***Process*** de VHDL.
  - ✓ Um *Process* é uma estrutura sintática que representa um processo especial, que permite descrever ações em sequência ao invés de concorrentemente (é um recurso de VHDL).
  - ✓ Outras construções são processos, tais como atribuições fora de um *Process* ou uma instância do comando *with*.
  - ✓ Alguns comandos, como o *for-generate* podem produzir múltiplos processos concorrentes.

PROCESSOS EM VHDL

➢ Exemplo:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity processos is
end processos;
architecture processos of processos is
  signal x, y, z, t: std_logic;
begin
  x <= not y;
  z <= x and t;
end processos ;
```

Esta descrição VHDL implementa um circuito com dois processos paralelos, cada um correspondendo a uma porta lógica (desenhe o circuito). Estes processos se comunicam?



O COMANDO *PROCESS* EM VHDL

- O comando ***process*** é uma construção em VHDL que serve para, entre outras utilidades.
  1. Descrever comportamento de um hardware de maneira sequencial, o que é complexo de realizar fora deste comando.
  2. Descrever hardware combinacional ou sequencial.
  3. Prover uma maneira natural de descrever estruturas mais abstratas de hardware tais como máquinas de estados finitas.
- Cada comando *process* corresponde a exatamente um processo (paralelo) VHDL.
- A semântica do comando *process* é complexa para quem não domina bem os modelos de funcionamento de hardware em geral, e de hardware síncrono em particular.

UM EXEMPLO DE COMANDO *PROCESS*

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity nand2_proc is
  port( a, b : in std_logic;
        c : out std_logic);
end nand2_proc;
architecture nand2_proc of nand2_proc is
begin
  process
    variable temp : std_logic;
  begin
    temp := not (a and b);
    if (temp='1') then
      c <= temp after 6ns;
    elsif (temp='0') then
      c <= temp after 5 ns;
    else
      c <= temp after 6ns;
    end if;
    wait on a,b;
  end process;
end nand2_proc;
```

Há vários conceitos a explorar:

- ✓ Variáveis.
- ✓ Se temp difere de 0 e de 1, o que vale?
- ✓ Semântica do comando *process*.
- ✓ O comando *wait*.
- ✓ Outras formas para descrever o mesmo comportamento:
  - Lista de Sensitividade.

UM EXEMPLO DE COMANDO *PROCESS*

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity nand2_proc is
  port( a, b : in std_logic;
        c : out std_logic);
end nand2_proc;
architecture nand2_proc of nand2_proc is
begin
  process(a, b)
    variable temp : std_logic;
  begin
    temp := not (a and b);
    if (temp='1') then
      c <= temp after 6ns;
    elsif (temp='0') then
      c <= temp after 5 ns;
    else
      c <= temp after 6ns;
    end if;
  end process;
end nand2_proc;
```

- Comandos *process* com um único comando *wait* antes do end process são muito comuns.
- VHDL provê sintaxe alternativa para estes, a lista de sensitividade.
  - ✓ Se tem a lista, *process* não pode ter *wait* e vice-versa.
  - ✓ Lista implica *wait on*, não wait for ou wait until ou outra forma de wait.

A SEMÂNTICA DO COMANDO *PROCESS*

- Uma maneira de compreender a semântica de um ***process*** é a partir de como funciona uma simulação VHDL deste comando.
  - ✓ Um ***process***, como qualquer processo em VHDL (e como qualquer pedaço de hardware), está eternamente em execução.
  - ✓ Dentro de um ***process***, a avaliação dos comandos é sequencial, ao contrário do que ocorre em VHDL fora de um ***process***, onde tudo é avaliado em paralelo.
  - ✓ Cada comando pode ter efeito sobre (atribuir novos valores a) sinais e/ou variáveis.
    - Atribuições a variáveis têm efeito imediato, como em programação.
    - Atribuições a sinais são projeções para o futuro (mesmo que o futuro seja agora!!!)
  - ✓ A definição de futuro é ditada pela execução do próximo comando *wait* na sequência de comandos do processo, seja este explícito ou implícito (quando se usa lista de sensitividade).

A SEMÂNTICA DE SIMULAÇÃO EM VHDL

- Lembrando: Simuladores VHDL são programas sequenciais executando em computadores. Contudo, a simulação deve refletir o comportamento paralelo do hardware, onde vários (talvez milhares) de processos operam simultaneamente.
- Algoritmo de um simulador VHDL: Um laço (talvez eterno) que:
  1. Com o tempo congelado no instante atual (começando em 0s), avalia cada processo existente, executando seus comandos até que se atinja um comando *wait* (explícito ou implícito). Quando isto ocorre, passa ao próximo processo, até que todos os processos tenham atingido um *wait* e suspendam.
     - ✓ Atribuições a sinais são colocadas em uma lista de eventos classificada pelo tempo (instante) em que estes devem ocorrer.
  2. Avança o tempo para o instante mais próximo em que algo deve ocorrer. Então, toma todos eventos projetados para este instante e os efetiva (ou seja, faz eles acontecerem!).
  3. Se a lista está vazia, a simulação para. Senão, volta ao passo 1.

PROBLEMAS COM O COMANDO *PROCESS*

- O problema de inferência de "*latches*" em comandos *process*:
  - ✓ Algumas descrições VHDL podem provocar a inferência de meios de armazenamento para garantir que um hardware se comporte da mesma forma que o modelo de simulação.
  - ✓ *Latches* inferidos costumam causar problemas. EVITEM ELES!!

Exemplo:

- A descrição VHDL ao lado escreve em B somente quando A vale "1010", e deve (pela semântica do *process*) manter o valor de B sempre que A for diferente deste valor. Assim, a síntese de hardware infere um latch para armazenar o valor de B.

```
entity gera_latch is
  port( A : in std_logic_vector(3 downto 0);
        B : out std_logic_vector(3 downto 0));
end gera_latch;
architecture gera_latch of gera_latch is
  signal int_B: std_logic_vector(3 downto 0):="0000";
begin
  B <= int_B;
  gera_latch: process (A)
  begin
    if (A="1010") then
      int_B<= not A;
    end if;
  end process;
end gera_latch;
```

REGISTRADOR

- Registradores são basicamente inferidos a partir de sinais declarados em processos com sinal de sincronismo (exemplo: *clock*). Para efeito de síntese e simulação, é aconselhável, embora não necessário introduzir um reset assíncrono,

```
process (clock, reset)
begin
  if reset = '1' then
    reg <= (others =>'0');
  elsif clock'event and clock='1' then
    reg <=  barramento_A;
  end if;
end process;
```

1. Como introduzir um sinal de "*enable*" no registrador, para habilitar a escrita?
2. Como implementar um registrador "*tri-state*" controlado por um sinal "*hab*"?

REGISTRADOR

- Exemplo de registrador com largura de palavra parametrizável e com habilitação (sinal "*ce*", do inglês *chip enable*):

generic

- ✓ Define um parâmetro para o módulo.
- ✓ Uso:

```
rx: regnbit generic map(8)
port map (ck  => ck,
          rst => rst,
          ce  => wen,
          D => RD,
          Q => reg);
```

```
entity regnbit is
  generic(N: integer := 16);
  port( ck, rst, ce : in std_logic;
        D : in std_logic_vector(N-1 downto 0);
        Q : out std_logic_vector(N-1 downto 0));
end regnbit;
architecture regnbit of regnbit is
begin
  process (ck, rst)
  begin
    if rst = '1' then
      Q <= others('0');
    elsif ck'event and ck = '0' then
      if ce = '1' then
        Q <= D;
      end if;
    end if;
  end process;
end regnbit;
```

REGISTRADOR

- Exemplo de uso de um registrador de deslocamento:

```
process (clock, reset)
begin
  if reset = '1' then
    A <= 0; B <= 0; C <= 0;
  elsif clock'event and clock='1' then
    A <= entrada;
    B <= A;
    C <= B;
  end if;
end process;
```

Exercício de fixação:

1. Desenhe o circuito acima utilizando flip-flops, supondo que A, B e C são sinais de 1 bit.
2. A ordem das atribuições (A,B,C) é importante ? O que ocorreria se fosse uma linguagem de programação tipo C?
3. Escreva o código para um registrador com deslocamento à esquerda e à direita.

REGISTRADOR

- Atribuição dentro e fora de *process*:

```
process (clock, reset)
begin
  if clock'event and clock='1' then
    A <= entrada;
    B <= A;
    C <= B;
    Y <= B and not C; -- dentro do process
  end if;
end process;
X <= B and not C; -- fora do process
```

- ✓ Qual a diferença de comportamento nas atribuições aos sinais X e Y?
  - Enquanto o sinal **X** é atualizado instantaneamente (assíncrono), o sinal **Y** só será atualizado na borda do clock (síncrono).

Conclusão:

- ✓ Sinais atribuídos em processos sob controle de um clock, serão sintetizados como saídas de flip-flops.
- ✓ Sinais fora de processos ou em processos sem variável de sincronismo (clock) serão, em geral, sintetizados como lógica combinacional.

MÁQUINA DE ESTADOS

```
entity MOORE is port(X, clock : in std_logic; Z: out std_logic); end;
architecture A of MOORE is
  type STATES is (S0, S1, S2, S3); -- tipo enumerado
  signal scurrent, snext : STATES;
begin
  controle: process(clock, reset)
  begin
    if reset='1' then
      scurrent <= S0;
    elsif clock'event and clock='1' then
      scurrent <= snext;
    end if;
  end process;
  combinacional: process(scurrent, X)
  begin
    case scurrent is
      when S0 => Z <= '0';
                 if X='0' then snext<=S0; else snext <= S2; end if;
      when S1 => Z <= '1';
                 if X='0' then snext<=S0; else snext <= S2; end if;
      when S2 => Z <= '1';
                 if X='0' then snext<=S2; else snext <= S3; end if;
      when S3 => Z <= '0';
                 if X='0' then snext<=S3; else snext <= S1; end if;
    end case;
  end process;
end A;
```

- Moore
  - ✓ Saídas são calculadas apenas a partir do ESTADO ATUAL.
- Exercício de fixação:
  1. Faça o diagrama de transição de estados desta máquina.
  2. Desenhe o circuito acima utilizando flip-flops e portas lógicas, supondo que X e Z são sinais de 1 bit.

ERROS COMUNS AO SE DESCREVER UM *PROCESS*

1. Duplo driver (fazer atribuições de um mesmo sinal em dois comandos *process* distintos).

✓ Errado

```
p1: process (clock, reset)
begin
  if reset='1' then
    X <= '0';
  elsif clock'event and clock='1' then
    if hab='0' or ctr='1' then
      X <= '1';
    end if;
  end if;
end process;
```

```
p2: process (clock)
begin
  if clock'event and clock='1' then
    if ctr='1' then
      X <= '1';
    end if;
  end if;
end process;
```

✓ Certo

```
p1: process (clock, reset)
begin
  if reset='1' then
    X <= '0';
  elsif clock'event and clock='1' then
    if hab='0' then
      X <= '1';
    end if;
  end if;
end process;
```

2. Lista de sensitividade incompleta.
3. Escrever código VHDL que cause inferência de "*latches*" (ver lâmina 12).
4. Realizar lógica junto com o teste de borda do sinal de clock.

✓ Errado

```
elsif clock'event and clock='1' and ce='0' then
```

✓ Certo

```
elsif clock'event and clock='1' then
    if ce='0' then
```

5. Não atentar para que sinais geram registradores (ver lâmina 16).

TRABALHO

PROJETO 1

1. Estude e diga que hardware o VHDL ao lado implementa.
2. Gere um testbench e simule este hardware.
3. Remova um dos sinais da lista de sensitividade do processo e mostre o que muda no comportamento do hardware, via nova simulação. Guarde ambos projetos para entregar.
4. Desenhe um diagrama de esquemáticos do circuito (c/ portas FFs, muxes, etc.).

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity prim_proc is
  port( in1, in2, in3, in4 : in std_logic;
        ctrl : in std_logic_vector (1 downto 0);
        sai : out std_logic);
end prim_proc;
architecture prim_proc of prim_proc is
begin
  process (in1, in2, in3, in4, ctrl)
  begin
    case ctrl is
      when "00"   => sai <= in1;
      when "01"   => sai <= in2;
      when "10"   => sai <= in3;
      when "11"   => sai <= in4;
      when others => null;
    end case;
   end process;
end prim_proc;
```

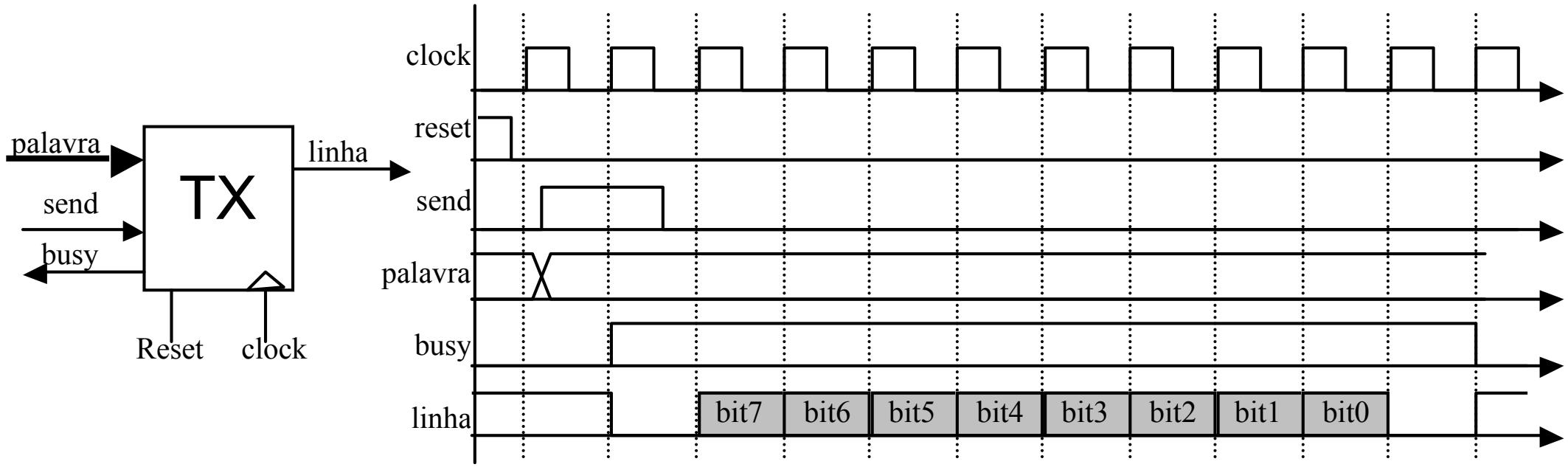
PROJETO 2

- Considere o circuito TX mostrado no próximo slide, o qual implementa uma transmissão serial de dados. A “linha” de dados (saída do circuito) está por default em ‘1’, indicando que não há transmissão de dados, e que a linhas está então em “repouso”. O protocolo de transmissão é o seguinte:

1. O mundo externo ao TX (test bench) coloca um byte em “palavra”, e sobe o sinal “send”, indicando ao módulo TX que há dado a ser enviado para a “linha”.
2. No primeiro ciclo de clock após a subida de “send” o módulo TX sobe o sinal de “busy”, impedindo que o mundo externo solicite novos dados. Concorrentemente a esta ação a linha sai do repouso, indo a 0 por um ciclo (bit denominado start bit).
3. Nos próximos 8 ciclos de clock o dado escrito em palavra é colocado, bit a bit, na “linha” serial.
4. No décimo ciclo de clock após a detecção do send a linha vai a zero (bit denominado stop bit) e o busy desce no final do ciclo.

PROJETO 2

- DICAS:
  - ✓ A máquina precisa ter no máximo 11 estados.
  - ✓ O controle da saída “linha” pode ficar dentro de um “process” combinacional.
  - ✓ O sinal de “busy” pode ser implementado como uma atribuição concorrente fora de comandos “process”.



PROJETO 2

- Para a validação, use o testbench abaixo:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity transmissor_tb is
end transmissor_tb;
architecture transmissor_tb of transmissor_tb is
  signal busy, linha, clock, reset, send: std_logic;
  signal palavra : std_logic_vector(7 downto 0);
begin
  UUT : entity work.transmissor
        port map (clock => clock, reset => reset, send => send,
                  palavra => palavra, busy => busy, linha => linha);
  process
  begin
    clock <= '1' after 5ns, '0' after 10ns;
    wait for 10ns;
  end process;
  reset   <= '1', '0' after 3 ns;
  send    <= '0', '1' after 23 ns, '0' after 50 ns, '1' after 160ns, '0' after 200 ns;
  palavra <= "11010001", "00100110" after 150ns;
end transmissor_tb;
```

RESUMO DO TRABALHO 2A

- O **Trabalho 2A** (T2A) consiste em um arquivo compactado (.zip) contendo:
  - ✓ Um relatório em PDF descrevendo os 2 projetos de acordo com o modelo apresentado.
  - ✓ Os fontes dos **DOIS** projetos (não entregar projeto ISE), contendo:
    - Os arquivos fontes de cada implementação.
    - Os arquivos de testbench de cada implementação.