ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO
Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos

PMR3303 Eletrônica Digital para Mecatrônica 1º semestre de 2019

Nome: _____ Nº USP: _____

# Experiência 4 DISPOSITIVO LÓGICO PROGRAMÁVEL – FPGA (REV. A)

A FPGA (*Field Programmable Gate Array*) é uma das tecnologias mais avançadas para a implementação de circuitos lógicos digitais. São dispositivos facilmente programáveis, de alta densidade, com os quais podemos implementar sistemas digitais equivalentes a milhões de transistores a um custo muito baixo.

Esta experiência é uma introdução ao ambiente de desenvolvimento Vivado da Xilinx para a programação de FPGAs. Além de conhecer os rudimentos da utilização deste software, vamos programar e testar uma FPGA usando a placa de desenvolvimento Basys 3 (Digilent, EUA). A sessão 9.6 do livro texto (Wakerly) explica resumidamente o que é uma FPGA e como funcionam. A FPGA e outros dispositivos programáveis mais antigos também são descritos na sessão 1.7.

A programação será feita em linguagem Verilog. Estude com ANTECEDÊNCIA o material introdutório sobre Verilog na sessão 5.4 do livro texto (Wakerly).

Além desta apostila, você vai precisar do manual de usuário da placa de desenvolvimento Basys 3 que usaremos no laboratório para testar nossos projetos em FPGA.

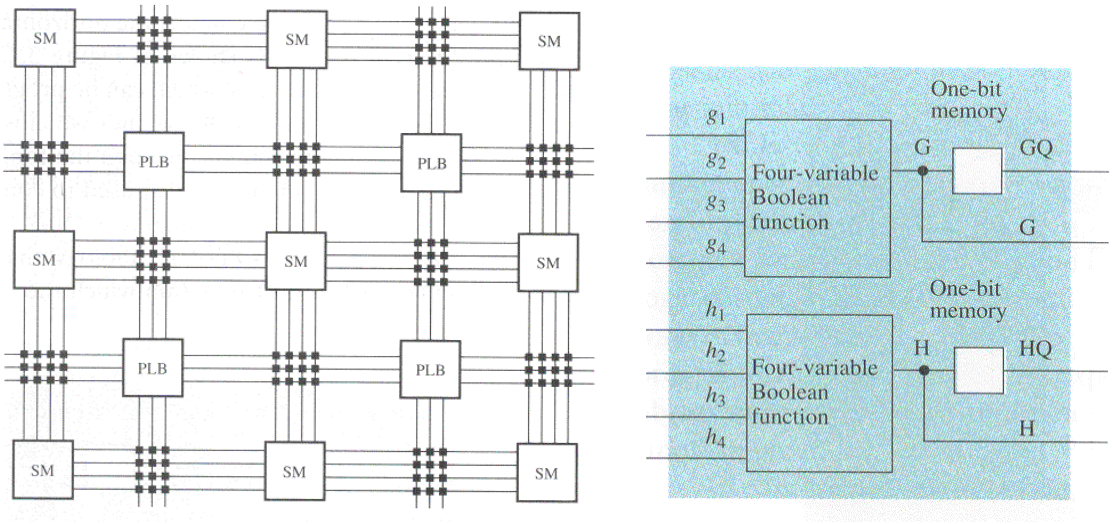
- Estude a apostila **com antecedência**. Sua compreensão será avaliada na aula por **ARGUIÇÃO ORAL**.
- Faça os **EXERCÍCIOS** contidos na apostila e tire dúvidas com os professores **com antecedência**.
- Traga para a aula a apostila **IMPRESSA**. Os pontos importantes devem estar **destacados ou grifados**.

## PARTE A TEORIA

### 4.1 FPGA (*Field Programmable Gate Array*)

Internamente, uma FPGA é composta por células padrão que podem ser programadas individualmente para constituir funções lógicas de *n* entradas e *m* saídas, chamadas PLB (*programmable logic block*). Matrizes de interconexões denominadas SM (*switch matrix*) também programáveis permitem conectar as saídas às entradas do próprio PLB (para formar circuitos realimentados) ou de outros PLBs. Como uma pastilha FPGA típica possui centenas de PLBs e SMs, temos um dispositivo capaz de implementar desde funções lógicas simples a sistemas digitais bastante complexos.

A Figura 4.1, ilustra a estrutura simplificada de uma FPGA. Neste exemplo, a PLB contém duas memórias programáveis (do tipo RAM), cada uma contendo 16 posições de 1 bit. Essa memória é chamada LUT (*look up table*), e cada uma delas permite implementar qualquer função booleana de 4 variáveis de entrada.




(a) (b)

Figura 4.1 a) Estrutura matricial de uma FPGA, com PLBs interconectadas por SMs; b) Exemplo de PLB, composta por duas LUTs de 4 variáveis (extraído de Gajski, "Principles of Digital Design")

Note que, neste exemplo, temos $n = 8$ e $m = 2$. Observe que a estrutura desse PLB não permite realizar qualquer função lógica de 8 variáveis, e sim duas funções de 4 variáveis cada. Além disso, a matriz de interconexão SM não permite conectar quaisquer PLBs. Mas estas restrições podem ser superadas, interconectando-se vários PLBs.

A Figura 4.2, ilustra como implementar um somador completo de 1 bit usando uma PLB. Repare que a LUT superior (G) reproduz a tabela da verdade do vai-um ($c_{i+1}$) e a inferior (H) contém a tabela da verdade do $i$-ésimo bit da soma ($s_i$). Como se tratam de funções de três variáveis, uma das entradas de cada LUT não é utilizada. No caso da LUT G, por exemplo, a entrada $g_1$ foi deixada em aberto, e portanto o conteúdo da LUT deve ser o mesmo tanto para $g_1 = 0$ como para $g_1 = 1$ (na figura, isto é indicado com X na coluna $g_1$ de endereços). Analogamente, na LUT H, a entrada $h_4$ não foi utilizada.

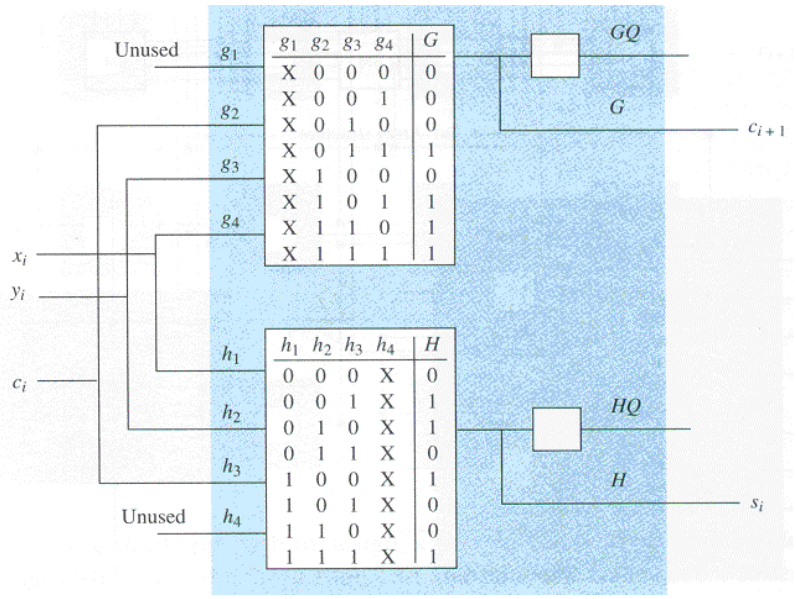



Figura 4.2 Somador completo ($c_{i+1}$ $s_i$) implementado com uma PLB
(extraído de Gajski, "Principles of Digital Design")

Uma pastilha FPGA possui vários pinos, e muitos deles podem ser configurados individualmente para servir de entradas ou de saídas para as funções programadas.

A programação de PLBs, SMs e pinos é armazenada internamente nas FPGAs em memória volátil, o que significa que uma pastilha pode ser reprogramada milhões de vezes. Isso as torna uma excelente tecnologia para a prototipagem rápida de novos sistemas digitais. Projetistas de circuitos integrados, por exemplo, costumam utilizá-las para testar pro parte o projeto de CIs complexos.

## 4.2 Programação

A programação interna de PLBs e SMs normalmente é bastante complexa, mesmo para se implementar funções lógicas bem simples. Para facilitar a sua vida, os fabricantes de FPGAs investem pesadamente no desenvolvimento de softwares integrados para a programação automática desses dispositivos. São os chamados *compiladores de silício*. A ideia é permitir que você faça o projeto utilizando portas lógicas e bibliotecas de funções, de forma hierárquica. O compilador cuida de converter seu projeto em sequências de bits que posteriormente são transferidas para a memória de configuração da FPGA.

Em linhas gerais, o ciclo de desenvolvimento com uma FPGA envolve as seguintes etapas.

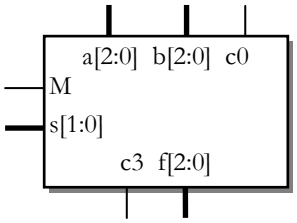
- Projeto digital (*design entry*): é o projeto propriamente dito do circuito que será implementado pela FPGA. Em projetos mais simples, podem ser usados editores gráficos para se desenhar o diagrama lógico do circuito – é o chamado modo de captura esquemática (*schematic entry*). Na prática, usam-se linguagens (*HDL, hardware description language*) para descrever os circuitos textualmente por meio de linhas de código que lembram programas de computador. Exemplos de linguagens: ABEL, VHDL e Verilog. Usaremos esta última no curso.
- Síntese: converte a descrição dos circuitos para uma linguagem intermediária mais simples e com menor nível de abstração, garantindo que o projeto possa ser fisicamente sintetizado.
- Simulação funcional: simula o funcionamento do circuito projetado com portas lógicas, sem considerar os atrasos de propagação, para que se testar a lógica implementada.
- Implementação: o projeto é compilado para formar os bits de programação da FPGA. É hora de se verificar se o modelo de FPGA a ser utilizada (*target*) comporta o projeto, isto é, se o CI contém elementos suficientes para implementar o circuito e se é rápida o suficiente para atender os requisitos de velocidade do projeto.

- Verificação: simula eletricamente o circuito implementado na FPGA, considerando tempos de atraso, para verificar se o circuito é capaz de operar na velocidade desejada, e se problemas como harzard não afetam o funcionamento do circuito.

Nesta experiência, utilizaremos o software Vivado 2016 versão Webpack, da Xilinx. Trata-se de um ambiente de desenvolvimento profissional bastante complexo. Nesta aula e na seguinte, procuraremos aprender a usar as funções elementares do programa.

## 4.3 Unidade Lógico-Aritmética

Como exemplo de aplicação, vamos implementar uma unidade lógico-aritimética (ULA) de 3 bits em FPGA no laboratório. A ULA implementa quatro funções lógicas e quatro operações aritméticas, conforme mostra a Figura 4.3. As entradas *a*[2:0] e *b*[2:0] são duas palavras de 3 bits, a saída *f*[2:0] de 3 bits é o resultado final, e $c_3$ é o *carry-out* gerado nas operações aritmética. As demais entradas *M* e *s*[1:0] selecionam a operação a ser efetuada. Na tabela, o índice "$C_2$" indica operação em notação complemento de 2, e *A* e *B* representam as palavras de entrada *a*[2:0] e *b*[2:0].




| | Funções | |
|---|---|---|
| $s_1$ $s_0$ | M = 0 (Modo Aritmético) | M = 1 (Modo Lógico) |
| 0 0 | A + B + c0 | A |
| 0 1 | A + 1 | A and B |
| 1 0 | $(A - 1)_{C2}$ | A or B |
| 1 1 | $(A - B - c0)_{C2}$ | not(A) |

Figura 4.3 Símbolo e operações da ULA

Para construí-la, vamos usar a arquitetura mostrado na Figura 4.4 (adaptada de "*Principles of Digital Design*", Gajski). O elemento central é um somador completo de três bits, representado pelo bloco FA3 (*Full Adder*). As parcelas x[2:0] e y[2:0] da soma são geradas pelos blocos LE3 (*Logic Extender*) e AE3 (*Aritmetic Extender*), enquanto que o vem-um c0E do somador é gerado pelo bloco CE (*Carry-in Extender*).

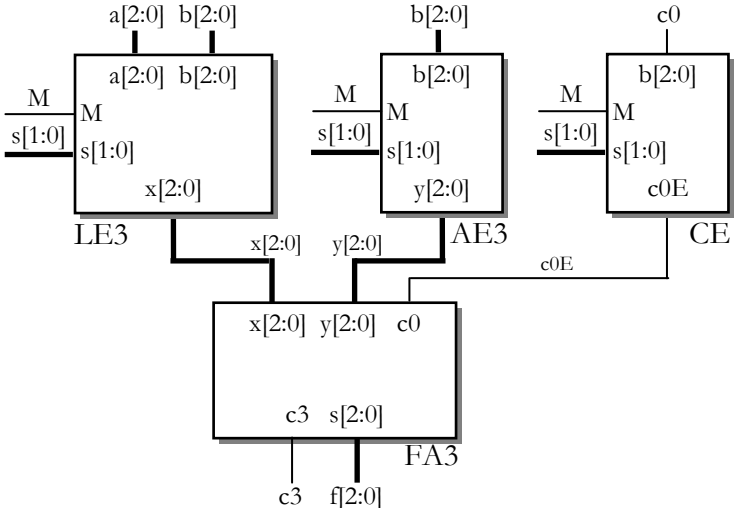



Figura 4.4 Arquitetura da ULA

A função dos extensores LE3, AE3 e CE é alterar as entradas do somador de modo a fazê-lo efetuar as operações desejadas. As saídas da ULA são as saídas do somador, e portanto

$$\{c3, f[2:0]\} = x[2:0] + y[2:0] + c0E\,, \qquad (4.1)$$

onde { c3, f[2:0] } representa uma palavra de 4 bits resultante da concatenação do bit c3 com os três bits de f[2:0], sendo c3 o bit mais significativo da palavra formada (essa notação é usada em Verilog).

### 4.3.1 Modo lógico

No modo lógico (com M = 1), a ideia é fazer com que extensor LE3 coloque em x[2:0] o resultado da operação lógica selecionada, e os blocos AE3 e CE zerem os sinais y[2:0] e c0E para que x[2:0] seja somado com zero e passe inalterado para as saídas da ULA. Ou seja, nas saídas da ULA tem-se

$$f[2:0] = x[2:0] + 000 + 0 = x[2:0] \quad \text{e} \quad c3 = 0\,.$$

Como o extensor LE3 faz operações lógicas bit a bit entre a[2:0] e b[2:0], podemos projetar a célula básica de um extensor lógico de um bit (LE1) e repeti-la três vezes, como mostra a Figura 4.6. Mas antes temos que ver também como o bloco LE3 deve funcionar no modo aritmético.

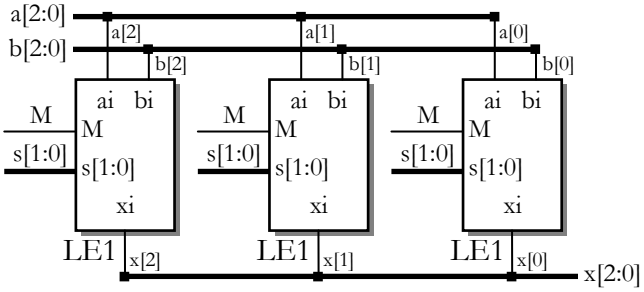



Figura 4.5 Estrutura interna do extensor lógico LE3

### 4.3.2 Modo aritmético

Repare que todas as operações aritméticas que a ULA efetua têm a entrada a[2:0] como primeiro operando. Ou seja, com M = 0, nas saídas da ULA temos somas do tipo

$$\{c3,\ f[2:0]\} = a[2:0] + y[2:0] + c0E\ .$$

Portanto o bloco LE3 deve copiar em x[2:0] os bits de a[2:0], e AE3 e CE devem modificar y[2:0] e c0E para que o somador efetue outras operações além da soma. A Tabela 4.1 mostra como implementar as operações aritméticas da ULA usando o somador.

Tabela 4.1 Funcionamento da ULA em modo aritmético (M=0).

| $s_1\ s_0$ | Função | y[2:0] | c0E | Soma efetuada | Observações |
|---|---|---|---|---|---|
| 0 0 | Adição | $b_2b_1b_0$ | c0 | $a_2a_1a_0 + b_2b_1b_0 + c0$ | Soma as entradas da ULA |
| 0 1 | Incremento | 000 | 1 | $a_2a_1a_0 + 000 + 1$ | Soma um usando a entrada de vem-um (c0) |
| 1 0 | Decremento | 111 | 0 | $a_2a_1a_0 + 111 + 0$ | 111 representa –1 em C2 com três bits |
| 1 1 | Subtração | $b_2'b_1'b_0'$ | c0' | $a_2a_1a_0 + b_2'b_1'b_0' + c0'$ | $(b_2'b_1'b_0' + c0')$ representa $(-b_2b_1b_0 - c0)$ em C2 |

De fato, as operações lógicas escolhidas para compor a ULA são tais que permitem construir o extensor AE3 também pela repetição da célula básica de um extensor aritmético de um bit (AE1), como mostra a Figura 4.6.

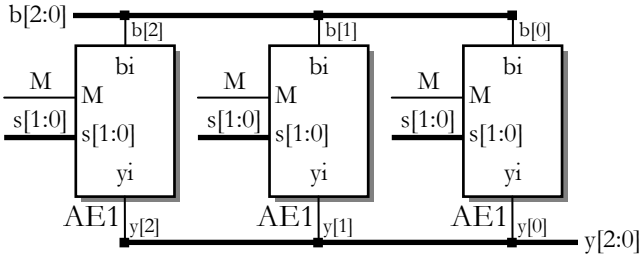



Figura 4.6 Estrutura interna do extensor aritmético AE3

### 4.3.3 Implementação dos extensores LE1, AE1 e CE

As conclusões anteriores estão condensadas na Tabela 4.2, que especifica as saídas que cada célula básica (de um bit) dos extensores deve gerar em função da operação a ser realizada.

Tabela 4.2 Definição das saídas xi, yi e c0E dos extensores LE1, AE1 e CE, respectivamente.

M = 0 (modo aritmético)

| $s_1\ s_0$ | Função | xi | yi | c0E |
|---|---|---|---|---|
| 0 0 | A+B+c0 | ai | bi | c0 |
| 0 1 | A+1 | ai | 0 | 1 |
| 1 0 | $(A-1)_{C2}$ | ai | 1 | 0 |
| 1 1 | $(A-B-c0)_{C2}$ | ai | bi' | c0' |

M = 1 (modo lógico)

| $s_1\ s_0$ | Função | xi | yi | c0E |
|---|---|---|---|---|
| 0 0 | A | ai | 0 | 0 |
| 0 1 | A and B | ai and bi | 0 | 0 |
| 1 0 | A or B | ai or bi | 0 | 0 |
| 1 1 | not(A) | ai' | 0 | 0 |

Assim, a conclusão do projeto da ULA se resume na implementação de três circuitos combinacionais de baixa complexidade, cujas expressões lógicas são dadas a seguir.

$$\textbf{LE1}: x_i = a_i.\overline{s_0} \; + \; \overline{M}.a_i + \; a_i.b_i.\overline{s_1} \; + \; M.\overline{a_i}.s_1.s_0 + \; M.\overline{a_i}.b_i.s_1 \tag{4.2}$$

$$\textbf{AE1}: y_i = \overline{M}.\left(\overline{b_i}.\mathrm{s}_1 \; + \; b_i.\overline{s_0}\right) \tag{4.3}$$

$$\textbf{CE}: c_0 E = \overline{M}.\left(\overline{c_0}.s_0 + c_0.\overline{s_1}\right) \tag{4.4}$$

Nota 1: necessitamos também de um somador binário (bloco FA3), mas não precisamos nos preocupar com ele. Como se trata de um circuito bem conhecido e bastante usado, poderemos implementá-lo facilmente na FPGA.

Nota 2: a verificação das expressões acima (usando mapas de Karnaugh, por exemplo) ficam por sua conta.

## 4.4 Verilog

Vamos dar a seguir alguns exemplos de como usar a linguagem Verilog para descrever os circuitos desta experiência. Não temos a pretensão de escrever um manual completo da linguagem, por isso complete seu estudo lendo o livro texto ou seu material de referência preferido.

### 4.4.1 Módulo LE3 – versão *assign*

A Figura 4.7 mostra como descrever o circuito do bloco LE3 usando atribuições do tipo *assign*. Palavras reservadas da linguagem Verilog são destacadas em lilás e em vermelho.

```
module LE3( input M,
            input [1:0] s,
            input [2:0] a, b,
            output [2:0] x );

    wire s1, s0;

    {s1, s0} = s;
    assign x[0] = (a[0] & ~s0) | (~M & a[0]) | (~s1 & a[0] & b[0]) | (M & s1 & s0 & ~a[0]) | (M & s1 & ~a[0] & b[0]);
    assign x[1] = (a[1] & ~s0) | (~M & a[1]) | (~s1 & a[1] & b[1]) | (M & s1 & s0 & ~a[1]) | (M & s1 & ~a[1] & b[1]);
    assign x[2] = (a[2] & ~s0) | (~M & a[2]) | (~s1 & a[2] & b[2]) | (M & s1 & s0 & ~a[2]) | (M & s1 & ~a[2] & b[2]);

endmodule // LE3_assign
```

Figura 4.7 Descrição do módulo LE3 com atribuições *assign*

Após a palavra reservada *module* e o nome de identificação do módulo (*LE3*), vem a lista de sinais de entrada e saída. NOTE: barramentos com mais de um bit tem a dimensão declarada ANTES do nome do sinal.

Em Verilog, sinais são conhecidos como *nets* e a linguagem define vários tipos de nets. As que usamos no módulo acima são do tipo ***wire*** : representam vias (ou trilhas) que conectam entradas e saídas de circuitos.

A palavra *assign* serve para atribuir um nome de uma *net* à saída de um circuito lógico.

Atribuições *assign* devem obrigatoriamente ser feitas a nets do tipo *wire*. Por *default*, o Verilog assume que *nets* de "input" e "output" dos módulos são do tipo *wire*.

A definição dos *wires* s1 e s0 não é de todo necessária, mas serve para melhorar a legibilidade do código e serve como exemplo de uso de *nets* auxiliares. A linha "{ s1, s0 }" indica que esses dois bits foram *concatenados* para receberem uma cópia dos bits de entrada s[1] e [0] em um único comando. Concatenações desse serão bastante usadas nos próximos exemplos.

### 4.4.2 Módulo LE3 – versão procedural

A descrição sinal à sinal mostrada na Figura 4.7 é no mínimo pouco interessante. E enfadonha.

Em Verilog, é possível descrever o funcionamento de um circuito em linguagem de nível mais alto. É a chamada forma *procedural*, que inclui comandos para instruir o compilador Verilog como construir o circuito.

Por exemplo, vê-se que os três bits do barramento *x* são geradas por circuitos idênticos, onde apenas se troca os bits de *a* e *b*. A Figura 4.8 mostra como implementar o *Logic Extender* de 3 bits inteiramente de forma procedural. Repare como o laço "*for*" descreve iterativamente a lógica dos três bits do módulo.

Um bloco procedural começa com ***"always @"***, seguido de uma lista de sinais que servem para ativar o bloco, chamada de *lista de sensibilidade*. De forma conveniente, é possível usar a forma "***(*)***" para indicar "todos os sinais de entrada que aparecem no bloco".

```
module LE3( input M,
            input [1:0] s,
            input [2:0] a, b,
            output [2:0] x );

    integer i;
    reg [2:0] xReg;
    reg s1, s0;

    always @ (*) begin
        {s1, s0} = s;
        for (i = 0; i <= 2; i=i+1) begin
            xReg[i] = (a[i] & ~s0) | (~M & a[i]) | (~s1 & a[i] & b[i]) | (M & s1 & s0 & ~a[i])
                | (M & s1 & ~a[i] & b[i]);
        end; // for i
     end // always

     assign x = xReg;

endmodule  // LE3
```

Figura 4.8 Descrição procedural do módulo LE3

Detalhe importante: apenas *nets* do tipo *reg* podem ser alteradas dentro de blocos procedurais (ou seja, aparecerem do lado esquerdo de igualdades). Por isso, desta vez s1 e s0 foram declarados como *reg*.

Outro detalhe: como vimos, sinais de entrada e saída de módulos são do tipo *wire* por default. Por isso criamos a net auxiliar *xReg*, do tipo *reg*, para receber as atribuições dentro bloco *always*. No fim, o valor de *xReg* é copiado na saída *x* (que é do tipo *wire*) por uma atribuição *assign*.

### 4.4.3 Geração de sinais de estímulo para simulação

O Verilog também serve para descrever como simular um circuito (o chamado *testbench*). A Figura 4.9 mostra um exemplo de *testbench* para o *Logic Extender* LE3. Ele contém a descrição do circuito a ser simulado (no caso, se resume à instanciação do módulo LE3) e gera os sinais de estímulo da simulação (sinais de entrada do módulo LE3).

Uma instanciação insere todo o circuito descrito em outro módulo, e no nosso exemplo ocorre na linha

`"LE3 UUT (M, s, a, b, x);"`

onde LE3 é o módulo instanciado, "UUT" (*Unit Under Test*) é um nome de identificação (equivalente aos U1, U2, etc. que usamos nos diagramas lógicos) e em seguida aparecem as *nets* conectadas às entradas e saídas do módulo.

A descrição dos sinais de estimulo é feita dentro de um bloco do tipo *initial*, que é do tipo **procedural.**

Algumas sutilezas desse exemplo:

- Usamos o inteiro *k* para varrer todas as possíveis combinações de 5 bits, de 00000 a 11111.
- As *nets* M, s[1:0], a[2:0] e b[2:0] são alteradas dentro do bloco "*initial*" e por isso precisam ser do tipo *reg*.
- O vetor x[2:0] precisa ser declarado como *wire*! Como está conectado à saída do módulo LE3 (e é alterado *fora* do bloco procedural por meio da instanciação de LE3), x deve ser uma *net* do tipo *wire*.
- A linha "**s = { k[3], k[2] };**" concatena dois bits da variável *k* para formar o vetor s[1:0].
- A construção "**{3{k[1]}}**" concatena 3 cópias do bit *k*[1] para formar um palavra de 3 bits, todos iguais a *k*[1].
- A construção "**#10**" faz o tempo de simulação avançar 10 unidades de tempo. A unidade de tempo usada é definida na primeira linha por ***'timescale 1ns / 1ps***. No caso, fará a simulação avançar 10 ns, com precisão de 0,001 ns nos cálculos de propagação.

Neste *testbench* temos também exemplos de comandos passados para o simulador.

- ***$timeformat*(-9, 0, "ns", 3)** Define a forma como o tempo de simulação é impresso. No caso, em $10^{-9}$ s, com 0 dígitos depois da vírgula, imprimindo "ns" como unidade e com 3 dígitos de comprimento. Se este comando não é usado, o tempo é impresso usando a precisão definida por "*timescale*" como unidade.
- ***$time*** Representa a variável interna do simulador que acumula o tempo de simulação.
- ***$write(...)*** Imprime uma mensagem na tela de console do simulador. O connteúdo "..." entre parêntesis segue as mesmas regras de formatação do comando "printf" da linguagem C.
- ***$finish*** Encerra a simulação.

No laboratório, faremos uma simulação funcional (ou *behavioral*). Esse tipo de simulação considera apenas a lógica do circuito, sem levar em conta os atrasos de propagação.

```
`timescale 1ns / 1ps    // Unidade de tempo e precisão da simulação
module LE3_sim( );      // Lista de entradas e saídas: vazia neste caso

    reg M;
    reg [1:0] s;
    reg [2:0] a, b;
    wire [2:0] x;

    integer k;

    LE3 UUT ( M, s, a, b, x );  // instanciação do módulo a ser simulado

    initial begin
        $timeformat(-9, 0, "ns", 3);  // formato para imprimir $time
        for (k = 0; k <= 31; k=k+1) begin
            M = k[4];
            s = { k[3], k[2] };
            a = {3{k[1]}};
            b = {3{k[0]}};
            #10;
            $write( "Tempo=%t: M=%b, s=%2b, a = %b, b=%b  =>  x=%b\n", $time, M, s, a, b, x);
        end; // for k
        $finish;
     end // initial

endmodule // LE3_sim
```

Figura 4.9 Simulação do módulo LE3.

### 4.4.4 Associando sinais à pinos da FPGA – *constraints*

Até agora, descrevemos de forma abstrata os sinais de circuitos. Mas para serem úteis na vida real, os sinais precisam estar conectados a pinos físicos do componente. No caso do Vivado, isso é feito por meio de um arquivo texto com formato próprio para isso. É o chamado arquivo de restrições (***constraints***), que tem extensão **"*.xdc*"**.

No caso do programa Vivado, cada pino de E/S é definido por um par de linhas no arquivo *.xdc*. Por exemplo: na placa Basys 3, a chave sw2 está conectada ao pino W16 da FPGA (confira no manual da placa), o que no arquivo .xdc ficaria

```
set_property PACKAGE_PIN W16 [get_ports sw2]
set_property IOSTANDARD LVCMOS33 [get_ports sw2]
```

São comandos próprios do programa Vivado que definem parâmetros internos para a implementação do circuito. Assim, "PACKAGE_PIN" é um parâmetro que define o pino que será conectado ao *net* sw2, o qual a função "`get_ports`" busca no módulo de topo do projeto. Já "LVCMOS33" indique que o pino será ligado a um *buffer* de E/S do tipo CMOS de 3,3 V.

### 4.4.5 Descrição em alto nível – módulo CE

Mesmo a versão procedural do módulo LE, mostrada na Figura 4.8, envolve um nível de abstração muito limitado. Isso porque descreve o circuito a partir de operações lógicas elementares, do tipo NOT, AND e OR.

Com linguagens poderosas como o Verilog, é possível ir muito além. A Figura 4.10 mostra um pequeno exemplo, usando o extensor de vem-um CE. Toda a descrição é feita de forma procedural, e descreve de forma quase direta a coluna c0E da Tabela 4.2, usando duas estruturas muito úteis do Verilog: ***if-else*** e ***case***.

A forma *if-else* é bem parecida com a de linguagens de programação como C.

Já o case tem uma particularidade que é importante destacar. Como se trata de uma descrição de um circuito físico, é conveniente que todos os casos possíveis do vetor *s* usado como seletor do *case* estejam previstos. Caso contrário, o circuito sintetizado pode não ser o esperado. Por isso, costuma-se sempre (e por segurança) incluir a linha "*default*" que será associada a qualquer caso não previsto.

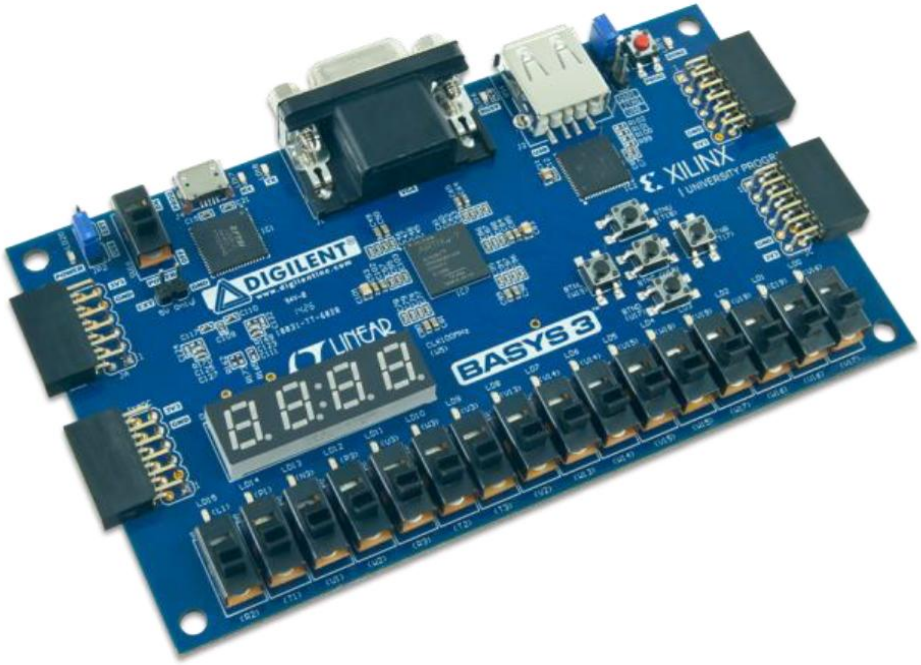
```
module CE(
    input M,
    input [1:0] s,
    input c0,
    output c0E
    );

    reg cReg;

    always @ (*) begin
        if (M)
            cReg = 0;
        else begin
            case ( s )
                2'b00 : cReg = c0;
                2'b01 : cReg = 1;
                2'b10 : cReg = 0;
                2'b11 : cReg = ~c0;
                default: cReg = 0;
            endcase;
        end; // else
    end // always

    assign c0E = cReg;

endmodule // CE
```

Figura 4.10 Descrição procedural em alto nível do módulo CE.

## 4.5 Placa de Desenvolvimento Basys 3

No laboratório, vamos testar os circuitos projetados utilizando a placa de desenvolvimento Basys 3, da Digilent. Ela contém uma FPGA modelo Artix 7, da Xilinx, de 236 pinos.

A placa contém ainda vários conectores e dispositivos que permitem verificar o funcionamento do circuito programado na FPGA. Dentre eles, temos 4 displays de 7 segmentos, 16 chaves de duas posições, 16 LEDs, 5 botões de contato momentâneo (*push buttons*).



Veja, em anexo, o manual de usuário da placa. Na página 15, a figura 16 indica em que pinos da FPGA os dispositivos estão conectados, e essa informação deve ser levada em conta no projeto do circuito. Por exemplo, para acender e apagar o led LD1, deve-se conectar o pino E19 da FPGA a uma saída do circuito projetado.

A placa já vem com um circuito de teste gravado em memória flash. Com o *Jumper* JP1 na posição 1-2, o circuito gravado é transferido para a FPGA ao se ligar a placa. O circuito de teste faz com que:

- Cada chave SW deve acionar o led LD correspondente
- O botão central deve apagar todos os displays e os demais um display de cada vez

O display conta sequencialmente

## 4.6 Pré-Relatório e Relatório

O formulário que se encontra na PARTE B da apostila constitui tanto o pré-relatório como o relatório desta experiência. Existem dois tipos de itens que você deverá responder:

- **Exercícios**: constituem o *pré-relatório* e **recomendamos fortemente** que sejam ser feitos com cuidado ***antes*** da aula. Se você tiver que fazê-los ou corrigi-los no laboratório, perderá tempo e poderá não conseguir concluir todas as atividades.
- **Anotações**: constituem o *relatório* e devem ser feitas individualmente *durante* a aula.

**ATENÇÃO: leia as atividades da PARTE B e não apenas os enunciados dos exercícios do pré-relatório**

Muitos detalhes necessários para fazer os exercícios estão descritos nas atividades em que se inserem. Além disso, você já terá uma noção do que deverá fazer e perderá menos tempo com a leitura durante a aula.

## 4.7 Outros módulos em Verilog usados nesta experiência

```
module xxAE3_top(
    input sw2,
    input sw1,
    input sw0,
    input sw15,
    output ld15
    );

    wire M;
    wire [1:0] s;
    wire [2:0] b, y;

    assign M = sw2;
    assign s = { sw1, sw0 };     // s[1]=sw1 e s[0]=sw0
    assign b = { 2'b0, sw15 };   // b[2]=b[1]=0 e b[0]=sw15
    assign ld15 = y[0];

    xxAE3 U1 ( M, s, b, y );     // Instanciação de xxAE3

endmodule // xxAE3_top
```

Figura 4.11 Módulo de topo para teste do *Aritmetic Extender* xxAE3 na placa

```
module ULA3(
    input M,
    input [1:0] s,
    input [2:0] a,
    input [2:0] b,
    input c0,
    output [2:0] f,
    output c3
    );

    wire [2:0] x, y;
    wire c0E;

    LE3 U1 (M, s, a, b, x);
    xxAE3 U2 (M, s, b, y);  // instanciação do seu módulo
    CE U3 (M, s, c0, c0E);

    assign {c3, f} = x + y + c0E;

endmodule // ULA3
```

Figura 4.12 Módulo ULA3.v

```
module ULA_top(
    input [15:0] sw,
    output [15:0] ld
    );

    wire M;
    wire [1:0] s;
    wire [2:0] a, b, f;
    wire c0, c3;

    assign M = sw[2];
    assign s = sw[1:0];
    assign a = sw[15:13];
    assign b = sw[11:9];
    assign c0 = sw[7];

    assign ld[3] = c3;
    assign ld[2:0] = f;
    assign {ld[15:13], ld[11:9], ld[7]} = 7'b1111111;
    assign {ld[12], ld[8], ld[6:4]} = 5'b00000;

    ULA3 U0 ( M, s, a, b, c0, f, c3 );  // Instanciação da ULA3

endmodule // ULA_top
```

Figura 4.13 Módulo ULA_top.v