Microprocessadores e Microcontroladores (27146)

Implementação de Processadores em Verilog

Prof. Ricardo Menotti (menotti@ufscar.br)

Atualizado em: 26 de abril de 2021

Departamento de Computação
Centro de Ciências Exatas e de Tecnologia
Universidade Federal de São Carlos

Conteúdo

A linguagem Verilog

A linguagem Verilog

- Verilog é uma linguagem complexa, mas neste curso não vamos abordar todas as suas potencialidades;
- O que vamos aprender será suficiente para projetar e testar uma grande variedade de circuitos, incluindo processadores;
- Iremos abordar as funcionalidades da linguagem a medida que avançarmos com os circuitos digitais;
- A principal habilidade desejada neste curso é a capacidade de traduzir com facilidade um circuito para Verilog e vice-versa;
- Isso só pode ser alcançado com a prática, pois assim como na programação, estudar problemas resolvidos não ajuda muito.

A linguagem Verilog

- Em Verilog há várias maneiras de se descrever um mesmo circuito, por exemplo, a partir:
  - **Funcional ou Lógica:** de funções ou portas básicas;
  - **Estrutural:** de uma hierarquia de componentes;
  - **Comportamental:** da descrição de seu comportamento;
- Pode-se usar combinações das metodologias.

Como NÃO escrever Verilog

- NÃO escrever código que se assemelhe a um programa de computador, contendo muitas variáveis e loops;
  - É difícil determinar qual circuito lógico as ferramentas CAD produzirão ao sintetizar código assim;
- Neste curso veremos exemplos completos de código Verilog que representam uma ampla gama de circuitos lógicos;
  - Neles o código é facilmente relacionado ao circuito lógico descrito;
  - Procure adotar o mesmo estilo de código;
- *Se não for possível determinar prontamente qual circuito lógico é descrito pelo código Verilog, então as ferramentas CAD provavelmente não sintetizarão o circuito que o projetista está tentando modelar;*
- **Analise o circuito resultante produzido pelas ferramentas de síntese;**

Exemplos de circuitos combinacionais

Exemplo: multiplexador

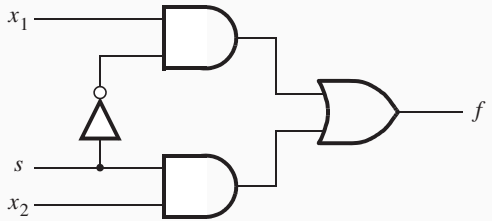


Figure 2.36 The logic circuit for a multiplexer.

```
1  // Behavioral specification
2  module example3(input x1, x2, s,
3                  output reg f);
4    always @(x1, x2, s)
5      if (s==0)
6        f = x1;
7      else
8        f = x2;
9  endmodule
```

```
1  module example2(input x1, x2, s, output f);
2    assign f = (x1 & ~s) | (x2 & s);
3  endmodule
```

```
1  module example1(x1, x2, s, f);
2    input x1, x2, s;
3    output f;
4
5    not (k, s);
6    and (g, k, x1);
7    and (h, s, x2);
8    or (f, g, h);
9  endmodule
```

```
1  module mux2to1 (w0, w1, s, f);
2    input w0, w1, s;
3    output f;
4    assign f = s ? w1 : w0;
5  endmodule
```

Exemplo: combinando funções lógicas

```
1  module example2 (x1, x2, x3, x4,
2                   f, g, h);
3    input x1, x2, x3, x4;
4    output f, g, h;
5
6    and (z1,  x1,  x3);
7    and (z2,  x2,  x4);
8    or  (g,   z1,  z2);
9    or  (z3,  x1, ~x3);
10   or  (z4, ~x2,  x4);
11   and (h,   z3,  z4);
12   or  (f,    g,   h);
13 endmodule
```

```
1  module example2 (x1, x2, x3, x4,
2                   f, g, h);
3    input x1, x2, x3, x4;
4    output f, g, h;
5
6    assign g = (x1 & x3)|(x2 & x4);
7    assign h = (x1 | ~x3)&(~x2 | x4);
8    assign f = g | h;
9  endmodule
```

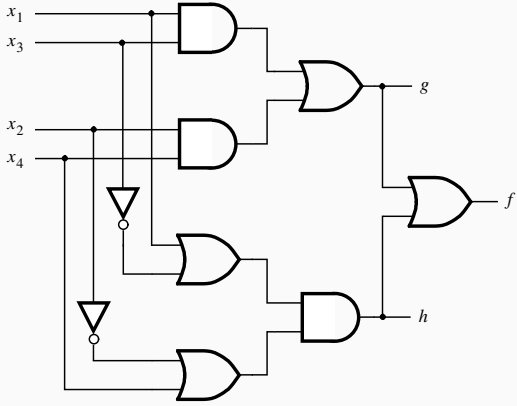


Figure 2.39 Logic circuit for the code in Figure 2.38.

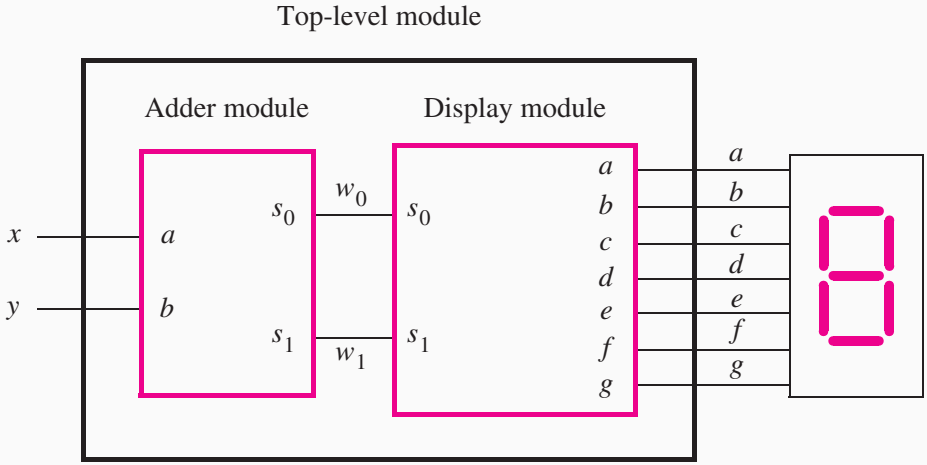


Figure 2.44 A logic circuit with two modules.

Hierarquia de componentes

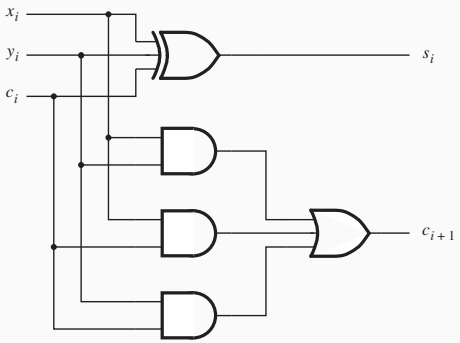
```
// Top-level module
module adder_display(x, y, a, b,
                     c, d, e, f, g);
  input x, y;
  output a, b, c, d, e, f, g;
  wire w1, w0;

  adder U1 (x, y, w1, w0);
  display U2 (w1, w0, a, b, c,
              d, e, f, g);
endmodule
```

```
// An adder module
module adder (a, b, s1, s0);
  input a, b;
  output s1, s0;

  assign s1 = a & b;
  assign s0 = a ^ b;
endmodule
```

```
// A module for driving a 7-segment display
↪  (0, 1 or 2)
module display(s1, s0, a, b,
               c, d, e, f, g);
  input s1, s0;
  output a, b, c, d, e, f, g;

  assign a = ~s0;
  assign b = 1;
  assign c = ~s1;
  assign d = ~s0;
  assign e = ~s0;
  assign f = ~s1 & ~s0;
  assign g = s1 & ~s0;
endmodule
```

Exemplos com aritmética computacional

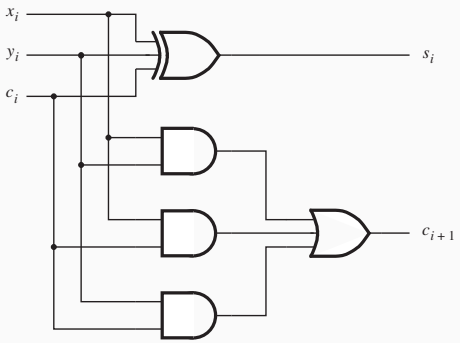
Exemplo: somador completo (full adder)



(c) Circuit

```
1  module fulladd (Cin, x, y, s, Cout);
2    input Cin, x, y;
3    output s, Cout;
4
5    xor (s, x, y, Cin);
6    and (z1, x, y);
7    and (z2, x, Cin);
8    and (z3, y, Cin);
9    or (Cout, z1, z2, z3);
10 endmodule
```

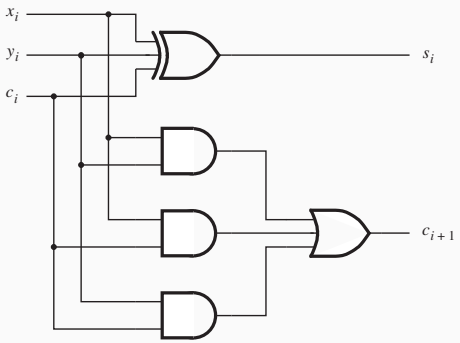
Exemplo: somador completo (sintaxe alternativa)



(c) Circuit

```
1  module fulladd (Cin, x, y, s, Cout);
2    input Cin, x, y;
3    output s, Cout;
4
5    xor (s, x, y, Cin);
6    and (z1, x, y),
7        (z2, x, Cin),
8        (z3, y, Cin);
9    or (Cout, z1, z2, z3);
10 endmodule
```

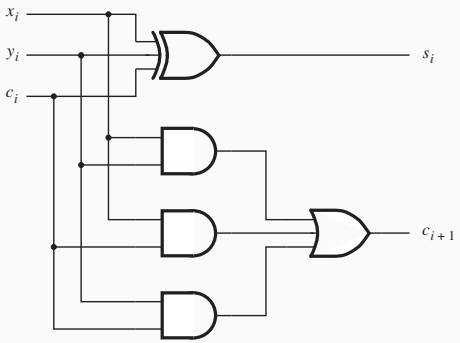
Exemplo: somador completo (atribuições contínuas)



(c) Circuit

```
1  module fulladd (Cin, x, y, s, Cout);
2    input Cin, x, y;
3    output s, Cout;
4
5    assign s = x ^ y ^ Cin;
6    assign Cout = (x & y) | (x & Cin) | (y &
     ↪  Cin);
7  endmodule
```

Exemplo: somador completo (sintaxe alternativa)



(c) Circuit

```
1  module fulladd (Cin, x, y, s, Cout);
2    input Cin, x, y;
3    output s, Cout;
4
5    assign s = x ^ y ^ Cin,
6           Cout = (x & y) | (x & Cin) | (y &
                ↪  Cin);
7  endmodule
```

Somador de 4 bits

```
1  module adder4 (carryin, x3, x2, x1, x0,
2                          y3, y2, y1, y0,
3                          s3, s2, s1, s0, carryout);
4    input carryin, x3, x2, x1, x0, y3, y2, y1, y0;
5    output s3, s2, s1, s0, carryout;
6
7    fulladd stage0 (carryin, x0, y0, s0, c1);
8    fulladd stage1 (c1, x1, y1, s1, c2);
9    fulladd stage2 (c2, x2, y2, s2, c3);
10   fulladd stage3 (c3, x3, y3, s3, carryout);
11 endmodule
```

Somador de 4 bits

```
1  module adder4 (carryin, x3, x2, x1, x0,
2                          y3, y2, y1, y0,
3                          s3, s2, s1, s0, carryout);
4    input carryin, x3, x2, x1, x0, y3, y2, y1, y0;
5    output s3, s2, s1, s0, carryout;
6
7    fulladd stage0 (.x(x0), .y(y0), .s(s0), .Cout(c1));
8    fulladd stage1 (c1, x1, y1, s1, c2);
9    fulladd stage2 (c2, x2, y2, s2, c3);
10   fulladd stage3 (c3, x3, y3, s3, carryout);
11 endmodule
```

Usando vetores

```
1  module adder4 (carryin, X, Y, S, carryout);
2    input carryin;
3    input [3:0] X, Y;
4    output [3:0] S;
5    output carryout;
6    wire [3:1] C;
7
8    fulladd stage0 (carryin, X[0], Y[0], S[0], C[1]);
9    fulladd stage1 (C[1], X[1], Y[1], S[1], C[2]);
10   fulladd stage2 (C[2], X[2], Y[2], S[2], C[3]);
11   fulladd stage3 (C[3], X[3], Y[3], S[3], carryout);
12 endmodule
```

Especificação parametrizada (genérica quanto ao tamanho)

```
1  module addern (carryin, X, Y, S, carryout);
2    parameter n = 32;
3    input carryin;
4    input [n-1:0] X, Y;
5    output [n-1:0] S;
6    output carryout;
7    reg[n-1:0]S;
8    reg carryout;
9    reg [n:0] C;
10   integer k;
11
12   always @(X or Y or carryin)
13   begin
14     C[0] = carryin;
15     for (k = 0; k < n; k = k+1)
16     begin
17       S[k] = X[k] ^ Y[k] ^ C[k];
18       C[k+1] = (X[k] & Y[k]) | (X[k] & C[k]) | (Y[k] & C[k]);
19     end
20     carryout = C[n];
21   end
22 endmodule
```

Usando atribuição aritmética

```
1  module addern (carryin, X, Y, S);
2    parameter n = 32;
3    input carryin;
4    input [n-1:0] X, Y;
5    output [n-1:0] S;
6    reg [n-1:0] S;
7
8    always @(X or Y or carryin)
9      S = X + Y + carryin;
10 endmodule
```

Calculando Carry-out e Overflow

```
1  module addern (carryin, X, Y, S, carryout, overflow);
2    parameter n = 32;
3    input carryin;
4    input [n-1:0] X, Y;
5    output [n-1:0] S;
6    output carryout, overflow;
7    reg[n-1:0]S;
8    reg carryout, overflow;
9
10   always @(X or Y or carryin)
11   begin
12     S = X + Y + carryin;
13     carryout=(X[n-1] & Y[n-1]) | (X[n-1] & S[n-1])
14                                | (Y[n-1] & S[n-1]);
15     overflow = carryout ^ X[n-1] ^ Y[n-1] ^ S[n-1];
16   end
17 endmodule
```

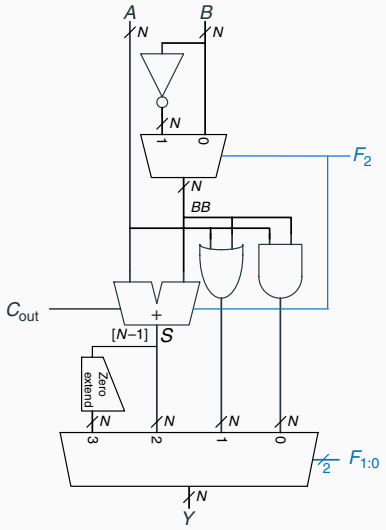
Sintaxe alternativa

```
1  module addern (carryin, X, Y, S, carryout, overflow);
2    parameter n = 32;
3    input carryin;
4    input [n-1:0] X, Y;
5    output [n-1:0] S;
6    output carryout, overflow;
7    reg [n-1:0] S;
8    reg carryout, overflow;
9
10   always @(X or Y or carryin)
11   begin
12     {carryout, S} = X + Y + carryin;
13     overflow = carryout ^ X[n-1] ^ Y[n-1] ^ S[n-1];
14   end
15 endmodule
```

Uma unidade lógica e artimética (ALU)

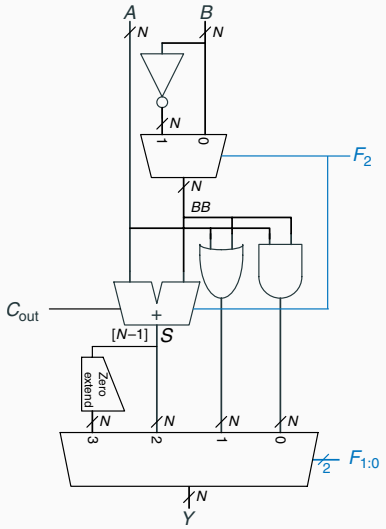
Table 5.1 ALU operations

$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A − B
111	SLT



Uma unidade lógica e artimética (ALU)

```
1  module alu(input  logic [31:0] a, b,
2             input  logic [ 2:0]  f,
3             output logic [31:0] y,
4             output logic zero);
5
6    logic [31:0] bb, b2, add_res, and_res, or_res, slt_res;
7
8    assign bb = ~b;
9    assign b2 = f[2] ? bb : b;
10   assign add_res = a + b2 + f[2]; // handle 2's complement
11   assign and_res = a & b2;
12   assign or_res  = a | b2;
13   assign slt_res = add_res[31] ? 32'b1 : 32'b0;
14
15   always_comb
16     case (f[1:0])
17       2'b00: y = and_res;
18       2'b01: y = or_res;
19       2'b10: y = add_res;
20       2'b11: y = slt_res;
21     endcase
22
23   assign zero = (y == 32'b0);
24 endmodule
```



Test bench

```
1  module talu();
2    logic clk;
3    logic [31:0] a, b, y, y_expected;
4    logic [ 2:0] f;
5    logic        zero, zero_expected;
6
7    logic [31:0] vectornum, errors;
8    logic [103:0] testvectors[10000:0];
9
10   alu dut(a, b, f, y, zero);
11
12   always begin
13     clk = 1; #50; clk = 0; #50;
14   end
15
16   initial begin
17     $readmemh("talu.tv", testvectors);
18     vectornum = 0; errors = 0;
19   end
```

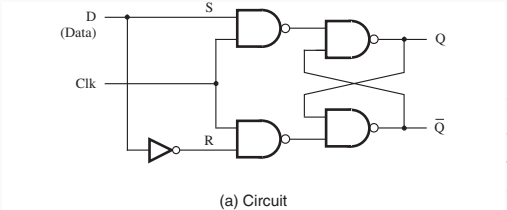
Test bench

```
21    always @(posedge clk)
22      begin
23        #1;
24        f = testvectors[vectornum][102:100];
25        a = testvectors[vectornum][99:68];
26        b = testvectors[vectornum][67:36];
27        y_expected = testvectors[vectornum][35:4];
28        zero_expected = testvectors[vectornum][0];
29      end
30
31    always @(negedge clk)
32      begin
33        if (y !== y_expected || zero !== zero_expected) begin
34          $display("Error in vector %d", vectornum);
35          $display(" Inputs : a = %h, b = %h, f = %b", a, b, f);
36          $display(" Outputs: y = %h (%h expected), zero = %h (%h expected)",
37            y, y_expected, zero, zero_expected);
38          errors = errors+1;
39        end
40        vectornum = vectornum + 1;
41        if (testvectors[vectornum][0] === 1'bx) begin
42          $display("%d tests completed with %d errors", vectornum, errors);
43          $stop;
44        end
45      end
46  endmodule
```

```
1   2_00000000_00000000_00000000_1
2   2_00000000_FFFFFFFF_FFFFFFFF_0
3   2_00000001_FFFFFFFF_00000000_1
4   2_000000FF_00000001_00000100_0
5   6_00000000_00000000_00000000_1
6   6_00000000_FFFFFFFF_00000001_0
7   6_00000001_00000001_00000000_1
8   6_00000100_00000001_000000FF_0
9   7_00000000_00000000_00000000_1
10  7_00000000_00000001_00000001_0
11  7_00000000_FFFFFFFF_00000000_1
12  7_00000001_00000000_00000000_1
13  7_FFFFFFFF_00000000_00000001_0
14  0_FFFFFFFF_FFFFFFFF_FFFFFFFF_0
15  0_FFFFFFFF_12345678_12345678_0
16  0_12345678_87654321_02244220_0
17  0_00000000_FFFFFFFF_00000000_1
18  1_FFFFFFFF_FFFFFFFF_FFFFFFFF_0
19  1_12345678_87654321_97755779_0
20  1_00000000_FFFFFFFF_FFFFFFFF_0
21  1_00000000_00000000_00000000_1
```

Exemplos com circuitos sequenciais

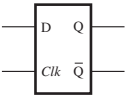
Latch D com enable



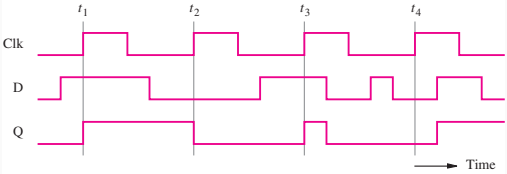
(a) Circuit

Clk	D	Q(t + 1)
0	x	Q(t)
1	0	0
1	1	1

(b) Truth table

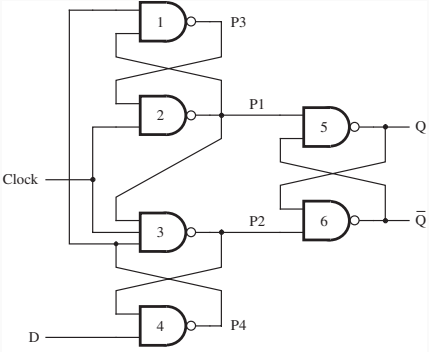


(c) Graphical symbol

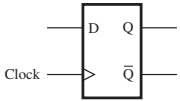



```
1  module D_latch (D, Clk, Q);
2    input D, Clk;
3    output Q;
4    reg Q;
5
6    always @(D or Clk)
7      if (Clk)
8        Q = D;
9  endmodule
```

Flip-flop D (sensível à borda)



(a) Circuit



(b) Graphical symbol

```
1  module flipflop(D, Clock, Q);
2    input D, Clock;
3    output Q;
4    reg Q;
5
6    always @(posedge Clock)
7      Q = D;
8  endmodule
```

Blocking

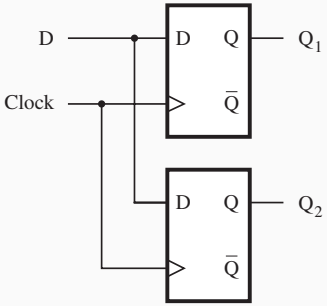


Figure 7.38 Circuit for Example 7.3.

```
1  module example7_3 (D, Clock, Q1, Q2);
2    input D, Clock;
3    output Q1, Q2;
4    reg Q1, Q2;
5
6    always @(posedge Clock)
7    begin
8      Q1 = D;
9      Q2 = Q1;
10   end
11 endmodule
```

Non-Blocking

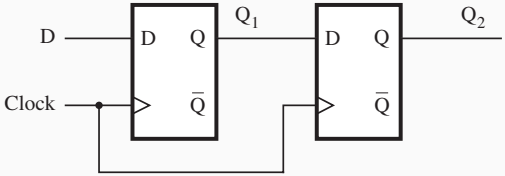
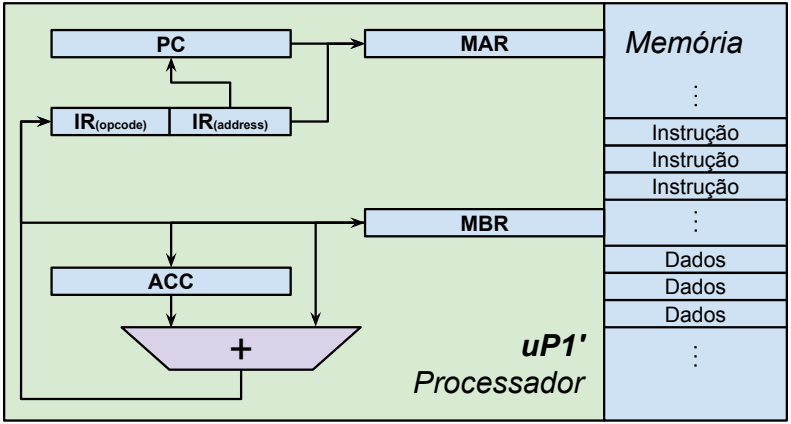


Figure 7.40 Circuit defined in Figure 7.39.

```
1  module example7_4 (D, Clock, Q1, Q2);
2    input D, Clock;
3    output Q1, Q2;
4    reg Q1, Q2;
5
6    always @(posedge Clock)
7    begin
8      Q1 <= D;
9      Q2 <= Q1;
10   end
11 endmodule
```

Um processador simples

Processador - uP1'




```
0   41   0100 0001   LOAD  A
1   52   0101 0010   ADD   B
2   33   0011 0011   STORE C
3   42   0100 0010   LOAD  B
4   31   0011 0001   STORE A
5   43   0100 0011   LOAD  C
6   32   0011 0010   STORE B
7   80   1000 0000   JUMP  0
    ...
240 FF   1111 1111   // Dados
241 00   0000 0000   // A
242 01   0000 0001   // B
243 00   0000 0000   // C
```

Experimente ele aqui!

Conjunto de instruções (ISA) do uP1' - 8 bits - 4 instruções - 2 formatos

- Formato M
  - `Endereço: 0x1111_endereço_D`

7	6	5	4	3	2	1	0	*bits*
0	0	0	0	endereço D				*não usada*
0	0	0	1	endereço D				*não usada*
0	0	1	0	endereço D				*não usada*
0	0	1	1	endereço D				STORE
0	1	0	0	endereço D				LOAD
0	1	0	1	endereço D				ADD
0	1	1	0	endereço D				*não usada*
0	1	1	1	endereço D				*não usada*

- Formato J
  - `Endereço: 0x0_endereço_I`

7	6	5	4	3	2	1	0	*bits*
1	endereço I (+7 bits)							JUMP

Organização de memória do uP1'

- Instruções

endereço								*palavras*
0	0	0	0	0	0	0	0	
				.				
				.				$2^7 = 128$
				.				
0	1	1	1	1	1	1	1	

- Não usada

endereço								*palavras*
1	0	0	0	0	0	0	0	
				.				$2^7 - 2^4 = 112$
1	1	1	0	1	1	1	1	

- Dados

endereço								*palavras*
1	1	1	1	0	0	0	0	
				.				$2^4 = 16$
1	1	1	1	1	1	1	1	

Processador - uP1'

```
1  module uP(
2    input clock, reset,
3    inout [7:0] mbr,
4    output logic we,
5    output logic [7:0] mar, pc, ir);
6
7    typedef enum logic [1:0] {FETCH, DECODE, EXECUTE}
     ↪  statetype;
8    statetype state, nextstate;
9
10   logic [7:0] acc;
11
12   always @(posedge clock or posedge reset)
13   begin
14     if (reset) begin
15       pc = 'b0;
16       state <= FETCH;
17     end
18     else begin
19       case(state)
20         FETCH: begin
21           we <= 0;
22           pc <= pc + 1;
23           mar <= pc;
24         end
25         DECODE: begin
26           ir = mbr;
27           mar <= {4'b1111, ir[3:0]};
28         end
29         EXECUTE: begin
30           if (ir[7] == 1'b1)            // jump
31             pc <= {1'b0, ir[6:0]};
32           else if (ir[7:4] == 4'b0100) // indirect load
33             acc <= mbr;
34           else if (ir[7:4] == 4'b0101) // add acc + data
35             acc <= acc + mbr;
36           else if (ir[7:4] == 4'b0011) // store
37             we <= 1'b1;
38         end
39         endcase
40         state <= nextstate;
41       end
42   end
```

Processador - uP1'

```
44    always_comb
45      casex(state)
46        FETCH:   nextstate = DECODE;
47        DECODE:  nextstate = EXECUTE;
48        EXECUTE: nextstate = FETCH;
49        default: nextstate = FETCH;
50      endcase
51
52    assign mbr = we ? acc : 'bz;
53  endmodule
```

```
56  module mem #(parameter filename = "ram.hex")
57            (input clock, we,
58             input [7:0] address,
59             inout [7:0] data);
60
61    logic [7:0] RAM[255:0];
62
63    initial
64      $readmemh(filename, RAM);
65
66    assign data = we ? 'bz : RAM[address];
67
68    always @(posedge clock)
69      if (we) RAM[address] <= data;
70  endmodule
```

Referências

Para saber mais e praticar...

- https://hdlbits.01xz.net/
- http://digitaljs.tilk.eu/
- http://hamblen.ece.gatech.edu/
- http://www.asic-world.com/verilog/
- https://www.edaplayground.com/x/sNSX

Bibliografia

- Brown, S. & Vranesic, Z. - Fundamentals of Digital Logic with Verilog Design, 3rd Ed., Mc Graw Hill, 2009
- Sarah L. Harris & David Money Harris - Digital Design and Computer Architecture, ARM Edition, Morgan Kaufmann, 2016
- J. Hamblen, T. Hall, and M. Furman, Rapid Prototyping of Digital Systems - SoPC Edition, Springer, August 2007.

Microprocessadores e Microcontroladores (27146)

Implementação de Processadores em Verilog

Prof. Ricardo Menotti (menotti@ufscar.br)

Atualizado em: 26 de abril de 2021

Departamento de Computação
Centro de Ciências Exatas e de Tecnologia
Universidade Federal de São Carlos