

# Projeto Final de Circuitos Digitais

Josué Rodrigues  
Instituto Federal do Ceará(IFCE)  
Av. Parque Central, 1315 - Distrito  
Industrial I, Maracanaú - Ceará,  
61939-140, Brazil

João Victor de Lima Pereira  
Instituto Federal do Ceará(IFCE)  
Av. Parque Central, 1315 - Distrito  
Industrial I, Maracanaú - Ceará,  
61939-140, Brazil

João Pedro Moreira  
Instituto Federal do Ceará(IFCE)  
Av. Parque Central, 1315 - Distrito  
Industrial I, Maracanaú - Ceará,  
61939-140, Brazil

**Resumo**— Este relatório aborda a implementação de um circuito fatorial utilizando a linguagem de descrição de hardware SystemVerilog. O circuito recebe um número de 8 bits e calcula seu valor fatorial, retornando o resultado em um valor de até 16 bits. Para isso, foram empregados métodos teóricos, diagramas ASMD e um código testado em uma bancada de testes. A validação incluiu uma série de vetores de teste gerados automaticamente por um script em Python. O circuito atendeu às expectativas de funcionamento, demonstrando uma implementação robusta e eficiente. Melhorias futuras poderão expandir a capacidade do circuito.

**Palavras-Chaves**— Circuito Fatorial, FSM, Diagrama ASMD

## I. INTRODUÇÃO

Em circuitos digitais, o termo máquina de estado, ou FSM, se refere a um circuito que sequencia um conjunto de estados predeterminados controlados por um clock e outros sinais de entrada[4], além disso, pode-se entender que o FSM descreve uma sequência de eventos discretos como  $t = 1, 2, 3, \dots$  etc[2]. Nesse sentido, a elaboração de um circuito fatorial será dado por implementação teórica e prática, utilizando softwares e linguagem de descrição de hardware.

## II. OBJETIVO

Implementar um circuito que receba um número de 8 bits e que devolva o número fatorial do número em 16 bits, que contenha os sinais **start**, **done** e **ready**.

## III. METODOLOGIA

Para desenvolver o projeto foi necessário estudos por meio de pesquisas e leitura de livros, como [2] e [4], além de fundamentações teóricas em [3] e análise de diagrama em [1]. Assim, para a implementação do circuito foi desenvolvido o diagrama com o caminho dos dados (Data Path), implementado em SystemVerilog, testado por meio de uma bancada de testes (Testbench) e de um arquivo com os vetores dos valores esperados da saída (testvectors), gerado com um código em Python.

## IV. RESULTADOS

### 1. Diagrama ASMD do Circuito Fatorial

Baseado nas exigências do objetivo e na lógica do circuito fatorial, conseguimos desenvolver o seguinte fluxograma, veja a figura 1. Note que, a lógica para o circuito, baseia-se na construção de quatro estados distintos, sendo, **IDLE**, **OP**, **DONE** e **ERROR**. O diagrama foi montado analisando os dados em [1], em que é fornecido um diagrama de estados, veja figura 2, além disso, estudamos as informações em [3], no qual é proposto um diagrama de estados para um circuito fatorial, baseado em

um método de algoritmo multiplicativo indiano. Dessa forma, optamos em seguir a linha lógica de [1], devido à familiaridade do método.

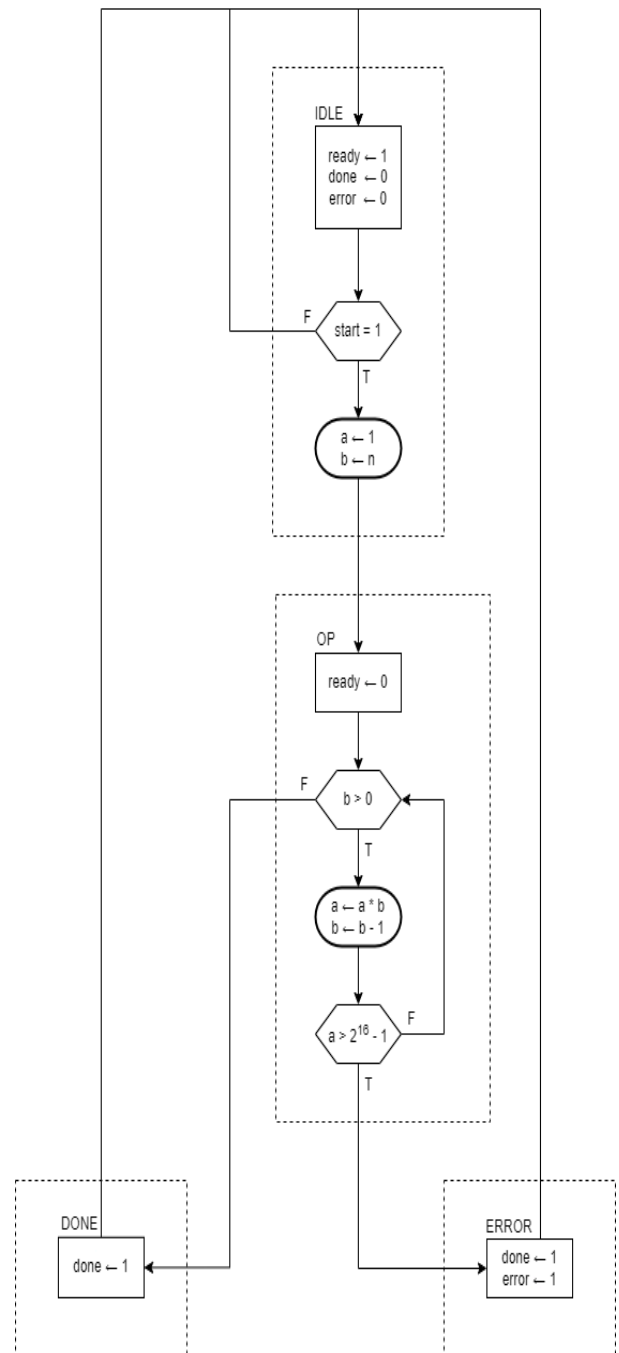


Figura 1. Diagrama ASMD do Circuito Fatorial.

Com o diagrama ASMD do circuito, partimos para a elaboração do diagrama de máquina de estado com caminho de dados. Veja a figura 3.

No diagrama temos, dois multiplexadores (MUXs), sendo o superior aquele que seleciona entre um valor constante de 1 e sua própria saída, passando o valor escolhido para o registrador a. Já o MUX inferior escolhe entre uma entrada externa N e a saída do registrador b, transmitindo o valor selecionado para o registrador b.

Assim, os registradores são responsáveis por armazenar dados temporariamente. O registrador guarda o valor que sai do MUX superior e utiliza esse valor em uma operação de multiplicação. Já o registrador b armazena o valor do MUX inferior e realiza uma operação de subtração, decrementando o valor em 1 a cada ciclo.

Além disso, o valor armazenado no registrador b é comparado com zero por uma unidade de comparação. Se o valor for igual a zero, essa unidade indica a condição. A FSM, por sua vez, controla o fluxo de dados através dos multiplexadores e monitora a comparação para decidir quando interromper o processo, que ocorre quando b chega a zero.

Por fim, o diagrama descreve um algoritmo onde um valor inicial é multiplicado por um número enquanto outro valor (b) é decrementado até atingir zero. A FSM gerencia a execução desse processo, ativando e desativando operações conforme necessário.

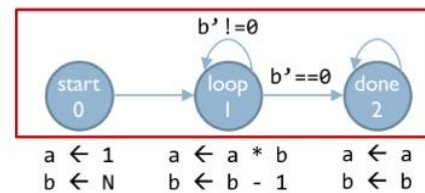


Figura 2. Diagrama de estados do Circuito Fatorial analisado.

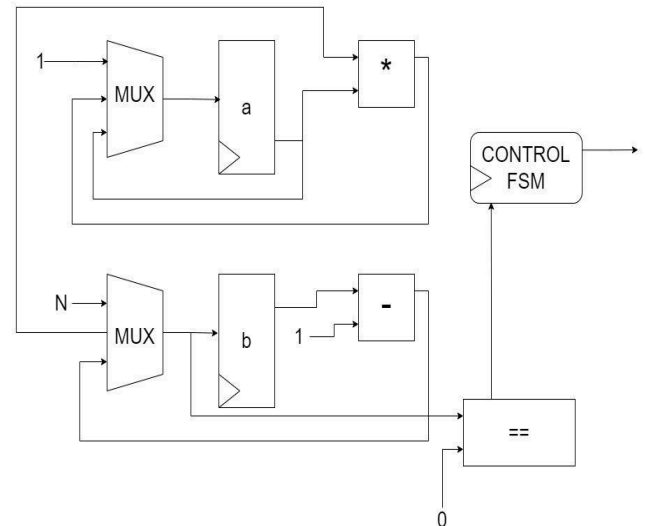


Figura 3. Diagrama da FSM com caminho de dados (Data Path).

## 2. Código em SystemVerilog.

Efetivamente, com o diagrama ASMD construído, o desenvolvimento do código em SystemVerilog torna-se simples. Veja a Figura 4.

Temos o módulo **factorial**, que possui os sinais de entrada: **clk**, **rst**, **start** e **n**. Como saídas, temos **out**, **ready**, **done** e **error**. O sinal **clk** é o clock que controla o circuito, **rst** é o reset, **n** é o número cujo fatorial queremos calcular, e **start** é o sinal que inicia a operação. Para as saídas temos, **out** que contém o resultado do fatorial, **ready** que indica se o circuito está pronto para iniciar, **done** que informa se a execução foi concluída e **error** que sinaliza se houve algum erro durante a execução. Acresça-se que o código define os estados do circuito usando **typedef enum {IDLE, OP, ERROR, DONE} state\_type**, refletindo o diagrama ASMD, com os estados **IDLE**, **OP**, **ERROR** e **DONE**. Esses estados controlam o fluxo de execução do cálculo. Além disso, o bloco **always\_ff** representa a lógica sequencial, responsável por armazenar e atualizar os valores do circuito a cada ciclo de clock. Nele, os estados do circuito e os valores dos registradores são atualizados conforme o estado corrente.

Já no bloco **always\_comb**, observa-se a implementação da lógica combinacional, responsável por determinar os valores dos registradores **a**, **b** e **state\_next** com base no estado atual do circuito. Esse bloco também define as saídas **ready**, **done** e **error**. No estado **IDLE**, o circuito define os valores iniciais de **a** (1) e **b** (n). Se o sinal **start** estiver ativo, o circuito avança para o estado **OP**, onde ocorre o cálculo do fatorial.

No estado **OP**, o circuito realiza as multiplicações sucessivas entre **a** e **b**, atualizando **a** com o resultado e decrementando **b** até que seu valor seja zero. Para garantir que não haja overflow, o circuito utiliza um registrador intermediário de 32 bits (**product**).

Caso o resultado ultrapasse o valor máximo de 16 bits, o circuito entra no estado **ERROR**, sinalizando o erro e retornando ao estado **IDLE**. Se o cálculo for bem-sucedido, o circuito transita para o estado **DONE**, que indica a conclusão da operação e, em seguida, retorna ao estado **IDLE** para aguardar um novo comando.

Por fim, a saída **out** contém o valor final do fatorial calculado, enquanto **ready** sinaliza que o circuito está pronto, **done** indica a conclusão e **error** aponta qualquer erro ocorrido durante a execução. Acesse [aqui](#) para visualizar o código em SystemVerilog do circuito.

```
1 module factorial(
2     input logic clk, rst,
3     input logic start,
4     input logic [7:0] n,
5     output logic [15:0] out,
6     output logic ready, done, error
7 );
8     typedef enum {IDLE, OP, ERROR, DONE} state_type;
9     state_type state_reg, state_next;
10
11     logic [15:0] a_reg, a_next;
12     logic [7:0] b_reg, b_next;
13     logic [31:0] product;
14
15     always_ff @(posedge clk, posedge rst) begin
16         if (rst) begin
17             state_reg <= IDLE;
18             a_reg <= 1;
19             b_reg <= 0;
20         end else begin
21             state_reg <= state_next;
22             a_reg <= a_next;
23             b_reg <= b_next;
24         end
25     end
26
27     always_comb begin
28         state_next = state_reg;
29         a_next = a_reg;
30         b_next = b_reg;
31         ready = 0;
32         done = 0;
33         error = 0;
34         product = 0;
35
36         case (state_reg)
37             IDLE: begin
38                 ready = 1;
39                 a_next = 1;
40                 b_next = n;
41                 if (start) begin
42                     state_next = OP;
43                 end
44             end
45
46             OP: begin
47                 if (b_next > 0) begin
48                     product = a_reg * b_reg;
49                     a_next = product[15:0];
50                     b_next = b_reg - 1;
51                     if (product > a_next) begin
52                         state_next = ERROR;
53                     end
54                 end else begin
55                     state_next = DONE;
56                 end
57             end
58
59             ERROR: begin
60                 done = 1;
61                 error = 1;
62                 state_next = IDLE;
63             end
64
65             DONE: begin
66                 done = 1;
67                 state_next = IDLE;
68             end
69
70             default: begin
71                 state_next = IDLE;
72             end
73         endcase
74     end
75
76     assign out = a_reg;
77 endmodule
78
```

Figura 4. Código do Circuito em SystemVerilog

### 3. Testbench.

Com o código em SystemVerilog já escrito, precisamos testá-lo e efetivar se a implementação foi um sucesso. Dessa forma, foi elaborada uma testbench para assegurar o comportamento correto do circuito ao calcular o fatorial de diferentes valores de entrada. Note que, no código, são definidos os sinais básicos que já estavam presentes no circuito, como o **clk** e o **rst**, além da inicialização do parâmetro **N**, que especifica a quantidade de casos de teste, e o array **testvectors**, que contém os valores de entrada e as saídas esperadas para cada teste. O registrador **expected\_out** é utilizado para armazenar a saída esperada correspondente a cada vetor de teste, facilitando a comparação com o resultado obtido pelo módulo em teste. E, em seguida, temos a instanciação do módulo do circuito, além da criação de uma lógica de **clock** usando "**always #5 clk = ~clk**", gerando um sinal de **clock** com período de 10 unidades de tempo.

```
1 module tb_factorial;
2   logic clk, rst;
3   logic start;
4   logic [7:0] n;
5   logic [15:0] out;
6   logic ready, done, error;
7
8   localparam N = 12;
9   logic [23:0] testvectors [N-1:0];
10  logic [15:0] expected_out;
11
12  factorial dut(clk, rst, start, n, out, ready, done, error);
13  always #5 clk = ~clk;
14
15  initial begin
16    $dumpfile("dump.vcd");
17    $dumpvars(0, tb_factorial);
18
19    clk = 1; start = 0; n = 0; expected_out = 0;
20    rst = 1; @(posedge clk); rst = 0;
21
22    $readmemb("testvectors.txt", testvectors, N-1, 0);
23    foreach (testvectors[i]) begin
24      {n, expected_out} = testvectors[i];
25      start = 1; @(posedge clk); start = 0;
26
27      wait(done);
28      assert(out == expected_out);
29      else $error(
30        "Erro no %do caso: out=%b expected_out=%b",
31        i+1, out, expected_out
32      );
33      @(posedge clk);
34    end
35
36    $finish;
37  end
38 endmodule
39
```

Figura 4. Código da Testbench do Circuito em SystemVerilog.

Já no bloco **initial**, temos a inicialização dos sinais e a ativação momentânea de um **reset**, para garantir que o circuito inicie em um estado conhecido, e em seguida o arquivo **testvectors.txt** é lido, carregando os valores de entrada e as saídas esperadas. Além disso, para cada vetor de teste, o valor de entrada **n** e a saída esperada são carregados, o circuito é ativado com **start**, e o sistema espera o sinal de conclusão **done**. Quando o cálculo termina, o valor de saída é comparado com o valor esperado usando uma asserção. Caso os resultados não coincidam, uma mensagem de erro é gerada com os detalhes do teste em questão.

Ao final de todos os testes, a simulação conclui com a confirmação de que o circuito está funcionando conforme o esperado. Acesse [aqui](#) para visualizar o código da Testbench.

### 4. Testvectors.

Agora, para concluir o desenvolvimento da testbench, precisamos gerar o arquivo que contém os valores de entrada e as saídas esperadas para cada teste, ou seja, o **testvectors**. Para isso, foi escrito em Python o seguinte código. Veja a figura 5.

```
1 WIDTH_IN      = 8
2 WIDTH_OUT     = 16
3 NUM_TESTCASES = 12
4
5 def to_bin(value, width):
6     return bin(value)[2:].zfill(width)
7
8 def calculate_factorial(value, width):
9     max_value = 2 ** width - 1
10    factorial = 1
11    for i in range(value, -1, -1):
12        factorial *= i if i else 1
13        if factorial > max_value:
14            break
15
16    truncated_factorial = factorial & max_value
17    return truncated_factorial
18
19 def main():
20     testvectors = []
21     for n in range(NUM_TESTCASES):
22         out = calculate_factorial(n, width = WIDTH_OUT)
23         n_to_bin = to_bin(n, width = WIDTH_IN)
24         out_to_bin = to_bin(out, width = WIDTH_OUT)
25         testvectors.append('.'.join([n_to_bin, out_to_bin]))
26
27     with open('testvectors.txt', 'w') as file:
28         file.write('\n'.join(testvectors))
29
30 if __name__ == '__main__':
31     main()
32
```

Figura 5. Código em Python responsável por gerar os Testvectors.


Na parte inicial do código temos, a definição das constantes **WIDTH\_IN**, **WIDTH\_OUT** e **NUM\_TESTCASES**, que especificam a número de bits da entrada, da saída, além do número total de casos de teste que serão gerados, respectivamente. Além disso, o código basicamente gira em torno da função **calculate\_factorial**, a qual é projetada para calcular o fatorial de um valor fornecido, considerando um limite de bits especificado. Primeiramente, define-se o valor máximo permitido (**max\_value**) com base na largura de bits fornecida. Em seguida, inicia-se o cálculo do fatorial com o valor inicial e itera de forma decrescente até zero, multiplicando o fatorial pelo valor atual da iteração.

Durante a multiplicação, há uma verificação para assegurar que o fatorial não exceda o valor máximo permitido. Se o fatorial ultrapassar esse limite, o cálculo é interrompido. Após completar a iteração, o fatorial é ajustado para garantir que não exceda o limite de largura de bits, por meio de uma operação Bitwise. O valor truncado é então retornado como o resultado da função.

Por fim, na função **main()**, o código gera os vetores de teste. Para cada valor de 0 até **NUM\_TESTCASES - 1**, o código calcula o fatorial, converte tanto o valor da entrada quanto o resultado para formato binário e os junta em uma string

formatada. Esses vetores de teste são então armazenados em uma lista. Finalmente, todos os vetores de teste são escritos em um arquivo chamado **testvectors.txt**, com cada vetor em uma nova linha, pronto para ser utilizado em testes de hardware ou simulações. Acesse [aqui](#) para visualizar o código de geração da Testvectors.

O resultado do arquivo **testvectors.txt** pode ser visualizado da seguinte forma. Veja a figura 6.



```

1  00000000_000000000000000001
2  00000001_000000000000000001
3  00000010_000000000000000010
4  00000011_0000000000000000110
5  00000100_000000000000011000
6  00000101_0000000001111000
7  00000110_0000001011010000
8  00000111_0001001110110000
9  00001000_1001110110000000
10 00001001_1100010011000000
11 00001010_0100111010100000
12 00001011_0001001101100000

```

Figura 6. Os valores armazenados no arquivo testvectors.txt.

Na imagem temos, a esquerda, a entrada esperada do circuito e, a direita, a saída esperada do circuito, sendo separados pelo caractere “\_”(underscore).

## 5. Forma de Onda.

Com tudo finalizado, utilizando o site **EDA PLAYGROUND**, partimos para a seguinte geração de da forma de onda. Veja a figura 7.

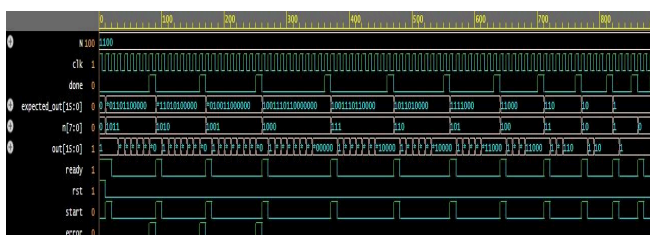


Figura 7. Forma de Onda Gerada do Circuito Fatorial.

Além disso, recomendamos, que acesse [aqui](#) para melhor visualização da forma de onda.

## V. CONCLUSÃO

Destarte, a implementação do circuito fatorial em SystemVerilog foi um sucesso. O projeto demonstrou a possibilidade de se calcular o fatorial de um número de 8 bits e retornar o resultado de no máximo 16 bits e que atenda os requisitos estabelecidos no objetivo. A validação foi efetivada de forma extensa ao se utilizar uma Testbench e uma testvectors, com geração automatizada de valores de testes por um código em Python. Além disso, os sinais de controle start, done e ready foram integrados e funcionam corretamente. O circuito é eficiente e válido, mostrando a importância de se estudar técnicas de implementação de circuitos para o desenvolvimento pessoal e profissional, como alunos de bacharelado em Ciência da Computação, para aprimorar o conhecimento acadêmico. Ademais, futuras melhorias podem incluir a otimização do circuito para reduzir o consumo de recursos e ampliar a capacidade de bits de entrada, para ser capaz de fazer a fatoração de números maiores. Por fim, a equipe está bastante satisfeita com o resultado apresentado.

## REFERÊNCIAS

- [1] CHEGG. Control FSM factorial b[0]: b[0] done start[0] loop[1]. Chegg, 2021. Disponível em: <https://www.chegg.com/homework-help/questions-and-answers/control-fsm-factorial-b-0-b-0-done-start-0-loop-1-2-draw-combinational-logic-transition-co-q89317229>. Acesso em: 11 set. 2024.
- [2] KUMAR, A. Anand. **Fundamentals of digital circuits**. PHI Learning Pvt. Ltd., 2016.
- [3] PANDA, Siba Kumar; SAHOO, Ankita; PANDA, Dhruba Charan. Design and Implementation of a Factorial Circuit for Binary Numbers: An AVM-Based VLSI Computing Approach. In: **Advanced Computing and Intelligent Engineering: Proceedings of ICACIE 2018, Volume 2**. Springer Singapore, 2020. p. 73-82.
- [4] Tocci, R. J., Widmer, N. S., and Moss, G. L. (2010). **Sistemas digitais**. Pearson Educacion. 22.v