Boas práticas de programação

Ítallo de Sousa Silva[1], Thiago Nascimento de Lima[2]

A premissa de tal documento é possibilitar ao leitor informar-se sobre boas condutas que podem possibilitar uma melhora na legibilidade de seu código. Abaixo, temos um exemplo de um código simples em SystemVerilog, faça uma rápida análise nele antes de prosseguir.

```
module top(input  logic clk_2,
           input  logic [7:0] SWI,
           output logic [7:0] LED,
           output logic [7:0] SEG,
           output logic [63:0] lcd_a, lcd_b,
           output logic [31:0] lcd_instruction,
           output logic [7:0] lcd_registrador [0:31],
           output logic [7:0] lcd_pc, lcd_SrcA, lcd_SrcB,
             lcd_ALUResult, lcd_Result, lcd_WriteData, lcd_ReadData,
           output logic lcd_MemWrite, lcd_Branch, lcd_MemtoReg, lcd_RegWrite);

        logic [2:0] a1, a2, a3;
        logic [3:0] a4;
        logic a5;
        always_comb begin
        a1 <= SWI[2:0];
        a2 <= SWI[4:3];
        a3 <= SWI[7:5];
        if (a2 == 0) begin
        a4 <= a1 + a3;
        a5 <= ((a1 + a3) > 'hf);
        end else if (a2 == 1) begin
        a4 <= a1 - a3;
        a5 <= (a1 < a3);
        end else if (a2 == 2) begin
        a4 <= a1 * a3;
        a5 <= ((a1 * a3) > 'hf);
        end else begin
        a4 <= a1 ^ a3;
        a5 <= 0;
        end
        LED <= a4;
        SEG[7] <= a5;
        end

endmodule
```

Qual sua opinião sobre o código? Ele está bem escrito? Você consegue entender o seu objetivo rapidamente? O código acima apresenta vários fatores que implicam num longo tempo para conseguir entendê-lo, agora imagine que esse mesmo modelo de código seja adotado para um código de centenas de linhas feito em equipe, consegue enxergar o problema que isso seria?

[1] itallo.silva@ccc.ufcg.edu.br
[2] thiago.lima@ccc.ufcg.edu.br

Nos dias de hoje é praticamente impossível que um software seja desenvolvido por uma única pessoa, uma vez que soluções cada vez mais complexas estão sendo requeridas pelos usuários. Devido a este fato, certas convenções foram sendo tomadas pelos profissionais de computação para que o desenvolvimento de software se dê de maneira mais ágil, prática e fácil de manter.

Em linguagens de descrição de hardware, como o SystemVerilog (exemplo acima) a adoção dessas convenções traz tantas vantagens quanto no desenvolvimento de software. Permitindo por exemplo, no âmbito acadêmico, um melhor entendimento da sua ideia por parte do professor. A seguir discutiremos, algumas práticas a serem adotadas nos seus códigos da disciplina (e da vida acadêmica e profissional, de modo geral).

No âmbito industrial, a falta da adoção dessas convenções pode levar a empresa à falência.

1. Indentação

É muito comum vermos iniciantes em programação criando seus algoritmos de forma desorganizada e desalinhada. Chamamos isso de código não indentado, tal má prática apresenta seus riscos, visto que, em linguagens como Python a forma em que se é feita indentação modifica escopos de código, alterando seu valor semântico. Em outros casos, como em SystemVerilog, a indentação não se faz obrigatória, uma vez que, esta possui um identificador de fim de linha (*ponto-e-vírgula*) e marcadores de início de blocos (*begin* e *end*). Entretanto, ainda em casos assim é recomendado a adoção de um padrão, para facilitar a visualização e entendimento do proposto, observe como podemos transformar um trecho do código acima e torná-lo mais simples de ler e entender.

```
always_comb begin
a1 <= SWI[2:0];
a2 <= SWI[4:3];
a3 <= SWI[7:5];
if (a2 == 0) begin
a4 <= a1 + a3;
a5 <= ((a1 + a3) > 'hf);
end else if (a2 == 1) begin
a4 <= a1 - a3;
a5 <= (a1 < a3);
end else if (a2 == 2) begin
a4 <= a1 * a3;
a5 <= ((a1 * a3) > 'hf);
end else begin
a4 <= a1 ^ a3;
a5 <= 0;
end
LED <= a4;
SEG[7] <= a5;
end
```

```
always_comb begin
        a1 <= SWI[2:0];
        a2 <= SWI[4:3];
        a3 <= SWI[7:5];

        if (a2 == 0) begin

                a4 <= a1 + a3;
                a5 <= ((a1 + a3) > 'hf);

        end else if (a2 == 1) begin

                a4 <= a1 - a3;
                a5 <= (a1 < a3);

        end else if (a2 == 2) begin

                a4 <= a1 * a3;
                a5 <= ((a1 * a3) > 'hf);

        end else begin

                a4 <= a1 ^ a3;
                a5 <= 0;

        end

        LED <= a4;
```

```
        SEG[7] <= a5;                                   end
```

Note que com essa simples mudança, o nosso código inicial já se torna mais simples de ser lido e entendido. Além de fazer uma boa indentação, procure não escrever linhas de código muito longas, isso também dificulta o entendimento.

Como sei se minha linha está muito longa? Um bom parâmetro para isto é se perguntar: eu preciso dar scroll pro lado no meu editor de texto para ver parte do meu código? Se sua resposta for sim, essas linhas que requerem deslocamento estão grandes. Se ainda acha isso muito subjetivo, adote um máximo de 100 caracteres por linha, e seja feliz.

2. Adote nomes significativos

Além de uma boa indentação, nomear as variáveis de maneira que elas sejam de fácil compreensão é necessário para permitir a um leitor externo entender ao que elas se referem, mesmo com pouco ou nenhum contexto direto do código. Por exemplo, no nosso código-exemplo podemos fazer a seguinte transformação:

```
always_comb begin

        a1 <= SWI[2:0];
        a2 <= SWI[4:3];
        a3 <= SWI[7:5];

        if (a2 == 0) begin

                a4 <= a1 + a3;
                a5 <= ((a1 + a3) > 'hf);

        end else if (a2 == 1) begin

                a4 <= a1 - a3;
                a5 <= (a1 < a3);

        end else if (a2 == 2) begin

                a4 <= a1 * a3;
                a5 <= ((a1 * a3) > 'hf);

        end else begin

                a4 <= a1 ^ a3;
                a5 <= 0;

        end

        LED <= a4;
        SEG[7] <= a5;

end
```

```
always_comb begin

  operando1 <= SWI[2:0];
  operacao <= SWI[4:3];
  operando2 <= SWI[7:5];

  if (operacao == 0) begin

          resultado <= operando1 + operando2;
          erro <= ((operando1 + operando2)>'hf);

  end else if (operacao == 1) begin

          resultado <= operando1 - operando2;
          erro <= (operando1 < operando2);

  end else if (operacao == 2) begin

          resultado <= operando1 * operando2;
          erro <= ((operando1 * operando2)>'hf);

  end else begin

          resultado <= operando1 ^ operando2;
          erro <= 0;

  end

  LED <= resultado;
  SEG[7] <= erro;

end
```

3. Evite números mágicos

Além de dar bons nomes para suas variáveis, é recomendado que você evite o uso de números soltos pelo código, isto é, números que não sejam facilmente decifráveis. Esses valores que aparecem magicamente no nosso código são os chamados *números mágicos*.

SystemVerilog nos permite solucionar esse problema dos números mágicos de uma forma simples, através da palavra reservada ***parameter*** (semelhante ao que C/C++ e Java permitem fazer com o modificador ***const***). Usando essa palavra podemos associar números a palavras, como faremos a seguir transformando nosso código:

```
parameter NINSTR_BITS = 32;
parameter NBITS_TOP = 8, NREGS_TOP = 32, NBITS_LCD = 64;
parameter NBITS_OP = 3, NBITS_RESULT = 4;
parameter OPERAND1_END = 2, OPERAND1_START = 0;
parameter OPERAND2_END = 7, OPERAND2_START = 5;
parameter OPERATION_END = 4, OPERATION_START = 3;
parameter ADD = 0, SUB = 1, MUL = 2, MAXVAL = 'hf;
parameter ERROR_EXIT = 7;
module top(input  logic clk_2,
           input  logic [NBITS_TOP-1:0] SWI,
           output logic [NBITS_TOP-1:0] LED,
           output logic [NBITS_TOP-1:0] SEG,
           output logic [NBITS_LCD-1:0] lcd_a, lcd_b,
           output logic [NINSTR_BITS-1:0] lcd_instruction,
           output logic [NBITS_TOP-1:0] lcd_registrador [0:NREGS_TOP-1],
           output logic [NBITS_TOP-1:0] lcd_pc, lcd_SrcA, lcd_SrcB,
             lcd_ALUResult, lcd_Result, lcd_WriteData, lcd_ReadData,
           output logic lcd_MemWrite, lcd_Branch, lcd_MemtoReg, lcd_RegWrite);

logic [NBITS_OP-1:0] operando1, operacao, operando2;
logic [NBITS_RESULT-1:0] resultado;
logic erro;

always_comb begin

        operando1 <= SWI[OPERAND1_END:OPERAND1_START];
        operacao <= SWI[OPERATION_END:OPERATION_START];
        operando2 <= SWI[OPERAND2_END:OPERAND2_START];

        if (operacao == ADD) begin

                resultado <= operando1 + operando2;
                erro <= ((operando1 + operando2)>MAXVAL);

        end else if (operacao == SUB) begin

                resultado <= operando1 - operando2;
                erro <= (operando1 < operando2);

        end else if (operacao == MUL) begin
```

```
                resultado <= operando1 * operando2;
                erro <= ((operando1 * operando2)>MAXVAL);

        end else begin

                resultado <= operando1 ^ operando2;
                erro <= 0;

        end

       LED <= resultado;
       SEG[ERROR_EXIT] <= erro;

  end

endmodule
```

Observe como a leitura do código agora é feita de forma bem mais simples, tendo em vista que todos os números estranhos que apareciam nele foram agora transformados em palavras significativas ao leitor.

4. Faça comentários

Por último, mas não menos importante, uma outra boa prática de desenvolvimento é adicionar comentários ao código. Comentários idealmente servem como um complemento para dizer ao leitor algo que o código não foi capaz de deixar explícito. Evite o emprego de comentários redundantes. Abaixo um exemplo do código acima, comentado.

```
parameter NINSTR_BITS = 32;
parameter NBITS_TOP = 8, NREGS_TOP = 32, NBITS_LCD = 64;
parameter NBITS_OP = 3, NBITS_RESULT = 4;
parameter OPERAND1_END = 2, OPERAND1_START = 0;
parameter OPERAND2_END = 7, OPERAND2_START = 5;
parameter OPERATION_END = 4, OPERATION_START = 3;
parameter ADD = 0, SUB = 1, MUL = 2, MAXVAL = 'hf;
parameter ERROR_EXIT = 7;
module top(input  logic clk_2,
           input  logic [NBITS_TOP-1:0] SWI,
           output logic [NBITS_TOP-1:0] LED,
           output logic [NBITS_TOP-1:0] SEG,
           output logic [NBITS_LCD-1:0] lcd_a, lcd_b,
           output logic [NINSTR_BITS-1:0] lcd_instruction,
           output logic [NBITS_TOP-1:0] lcd_registrador [0:NREGS_TOP-1],
           output logic [NBITS_TOP-1:0] lcd_pc, lcd_SrcA, lcd_SrcB,
             lcd_ALUResult, lcd_Result, lcd_WriteData, lcd_ReadData,
           output logic lcd_MemWrite, lcd_Branch, lcd_MemtoReg, lcd_RegWrite);

//Implementação de uma ULA de inteiros, com 4 operações.
```

```
//As operações possíveis são adição, subtração, multiplicação e ou-exclusivo.

logic [NBITS_OP-1:0] operando1, operacao, operando2;
logic [NBITS_RESULT-1:0] resultado;
logic erro;

always_comb begin

        //Entrada de dados
       operando1 <= SWI[OPERAND1_END:OPERAND1_START];
        operacao <= SWI[OPERATION_END:OPERATION_START];
        operando2 <= SWI[OPERAND2_END:OPERAND2_START];

        if (operacao == ADD) begin

             //Realiza a operação de adição
               //Gera erro se o valor da soma exceder o valor máximo, 15 em decimal
                resultado <= operando1 + operando2;
                erro <= ((operando1 + operando2)>MAXVAL);

        end else if (operacao == SUB) begin

             //Realiza a operação de subtração
               //Gera erro se o resultado da operação for negativo
                resultado <= operando1 - operando2;
                erro <= (operando1 < operando2);

        end else if (operacao == MUL) begin

             //Realiza a operação de multiplicação
               //Gera erro se o valor do resultado exceder o valor máximo, 15 em decimal
                resultado <= operando1 * operando2;
                erro <= ((operando1 * operando2)>MAXVAL);

        end else begin

             //Realiza a operação de ou-exclusivo.
               //Não pode gerar erros.
                resultado <= operando1 ^ operando2;
                erro <= 0;

        end

       //Saída de dados
       LED <= resultado;
       SEG[ERROR_EXIT] <= erro;

  end

endmodule
```

Como mostrado para inserir um comentário basta iniciar com //, outra alternativa é a criação de blocos de comentários, que pode ser feita da seguinte forma: ***/* bloco de comentário */***.

5. Considerações finais

Por fim, é válido ressaltar que localizar erros ou manter um código mal escrito, demandará muito mais tempo e esforço do que um código bem estruturado que respeita as regras básicas para organização de código, anteriormente apresentadas. O tempo gasto para padronizar para boas práticas se pagará com o passar do tempo e o crescer do código.

Cabe ainda dizer, que apesar de termos utilizado SystemVerilog como linguagem de exemplo, essas práticas são válidas para todas as linguagens, cabendo ao programador buscar as ferramentas que a linguagem oferece para ajudá-lo a seguir tais regras.

Referências

https://www.devmedia.com.br/boas-praticas-de-programacao/31163
https://www.devmedia.com.br/boas-praticas-de-programacao/21137
Robert Cecil Martin, "Clean Code", ISBN 9780132350884