Introdução a Sistemas Lógicos - 2025/1

Rafael Augusto Resende Miranda

Cifra XOR - FSM

---

1 Descrição sucinta do funcionamento do projeto

Entradas do circuito:

1. **clk:** sinal de clock;
2. **reset:** reset síncrono;
3. **start:** sinal para iniciar o processo de cifragem;
4. **key[7:0]:** chave com a qual se faz a criptografia de 8 bits;
5. **plaitext[7 :0]:** texto claro de 8 bits.

Saídas do circuito:

1. **done:** sinal que indica a conclusão da operação;
2. **ciphertext[7:0]:** texto de 8 bits após a cifra.

```
module cifra_xor(
    input clk,
    input reset,
    input start,
    input [7:0] plaintext,
    input [7:0] key,
    output reg [7:0] ciphertext,
    output reg done
);
```

Figure 1: Inputs e Outputs

Existem também algumas reg que armazenam quais são os estados atuais/próximos estados e um contador, para que seja possível realizar a operação bit a bit nas palavras de 8 bits.

Quando o reset é ativado, o sistema entra em condição inicial e sofre um reset geral. É assim que iniciamos o nosso circuito, com reset = 1. Logo em segudia, voltamos reset para 0 novamente.

O circuito criado é uma Máquina de Estado Finito (FSM) de Mealy, em que a saída depende da entrada. Ela é composta por 4 estados principais: IDLE, LOAD, PROCESS e DONE:

1. **IDLE:** além de ser o estado inicial do circuito, é o estado padrão onde o sistema irá se encontrar, ou seja, caso não haja nenhuma entrada, o circuito estará em IDLE. A condição para sair desse estado é que a entrada start seja 1. Dessa forma, o programa dá início e vai para o estado de LOAD. Nesse estado, o done é zerado;

2. **LOAD:** o estado LOAD serve como um estado intermediário entre o IDLE e o PROCESS, garantindo que o sinal de começar fique separado por um estado do de processamento. No LOAD, o ciphertext e o contador são zerados, para evitar possíveis conflitos em situações em que mais de uma palavra será cifrada;

3. **PROCESS:** nesse estado ocorre a cifragem XOR de um bit por vez, em que um contador que vai de 0 até 7, ou seja, percorre todos os bits da palavra, serve de index para acessar o bit certo. Enquanto o valor do contador for menor que 7, o contador é incrementado e o estado não se altera. A cifragem é realizada e já é destinada ao seu exato lugar no ciphertext, também utilizando a indexação do contador. Assim que condição de (contador $<$ 3'd7) não for cumprida, o circuito vai para o estado DONE, onde irá finalizar.

4. **DONE:** aqui, o nosso circuito já realizou a cifragem completa da palavra e sinaliza definindo a saída done = 1. Após isso retorna para IDLE, e lá o circuito fica pronto para receber novas palavras/chaves.

2 Justificativa das decisões de projeto.

Nesse projeto, optei por utilizar a FSM de Mealy, já que o ciphertext, a saída principal, depende das entradas key e plaintext, que resultam na palavra cifrada. Logo, me pareceu o ideal, posto que nas máquinas de Mealy, a saída depende da entrada e dos estados atuais.

A respeito do tempo, defini um tempo de clock de 10ns, onde a cada 5ns ele inverte de valor. Em certos momentos do código, decidi deixar alguns "ciclos de segurança", para evitar race conditions e também garantir que os valores sejam lidos, propagados e que contaminem o circuito corretamente. Por exemplo, como as palavras têm tamanho 8, o mínimo de ciclos de clock para que todos os bits

```
initial begin
   $dumpfile("cifra_xor_waves.vcd");
   $dumpvars(0, dut);
    clk = 1'b0;
  forever #(tempo_clk/2) clk = ~clk;
end
```

```
      #(tempo_clk);
        start      = 1'b1;
      #(tempo_clk * 11);
```

da palavra sejam cifrados é 8, logo, decidi deixar ciclos para garantir que toda a palavra seja cifrada e que a troca de estados ocorra como esperado.

Outras escolhas que tive que fazer foi a de representação de estados, e eu o fiz utilizando "encoding", assim como vimos nas aulas. Como temos somente 4 estados, é possível cobrir todas eles com 2 bits: $2^2 = 4$.

O contador de 3 bits foi uma saída simples que pensei para conseguir conciliar as operações entre a key, o plaintext e o ciphertext, já que todas possuem o mesmo tamanho. Como ele possui 3 bits, é possível contar de 0 até 7, que é exatamente o que eu precisava. Isso fez com que a dinâmica das operações ficasse mais fácil.

Em relação aos estados, o programa, da forma que foi proposto, exigia o uso de 4 estados, mas acredito que seria possível realizá-lo sem o estado de LOAD. No entanto, o estado de LOAD pode auxiliar a escalar o projeto para tamanhos diferentes de plaintext e chave, além de permitir que mais de uma palavra seja

```
reg [1:0] current_state, next_state;

parameter IDLE    = 2'b00,
          LOAD    = 2'b01,
          PROCESS = 2'b10,
          DONE    = 2'b11;

reg [2:0] contador;
```

Figure 2: Encoding dos estados e contador

cifrada sem que seja necessário utilizar o reset síncrono.

Além disso, optei também pela ativação do circuito em bordas ascendentes do clock (flip-flop ascendente), já que é o modelo mais comum e o que nós vimos durante as aulas. Para isso, usamos o always @(posedge) na lógica sequencial. Para a lógica combinacional, usei o always @(*), já que não depende de um clock.

Agora, a respeito dos testes, realizei 4 testes, nos quais os 4 passaram. O 1° e o 2° são uma operação inteira da cifra, onde o plaintext é criptografado com uma key e logo em seguida, descriptografado com a mesma chave. Como esperado, o resultado ciphertext do teste 2 é igual ao plaintext do teste 1. Pela lógica da CIFRA XOR, está correto. Os outros testes são testes simples para conferir se a criptografia está funcionando de acordo com o esperado.

3 Diagrama da FSM implementada.

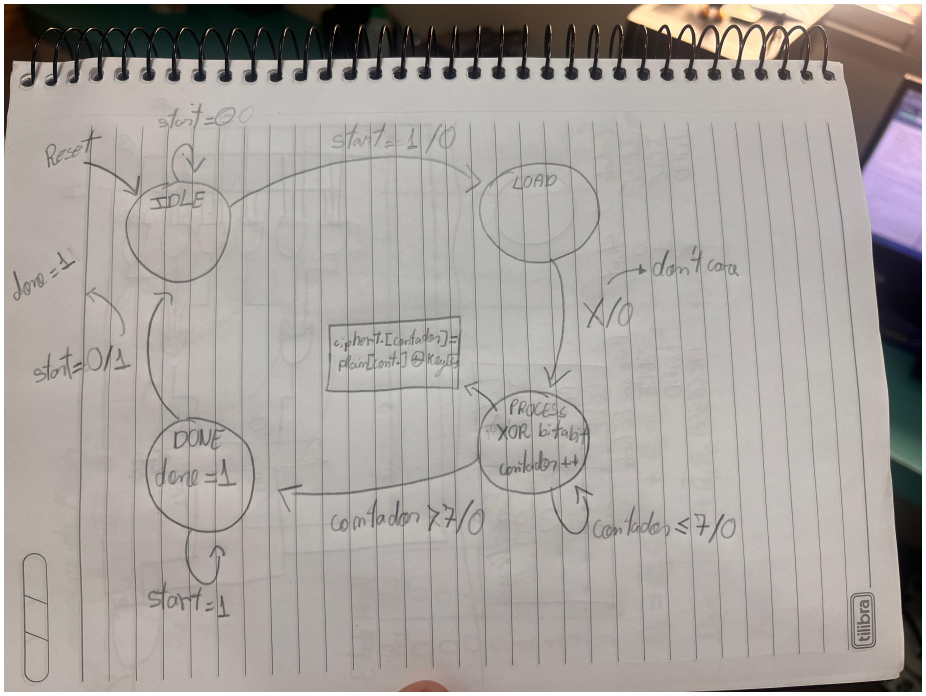

Figure 3: Diagrama da FSM implementada

4 Formas de onda geradas (diagrama de tempo) a partir dos testes realizados no testbench.

[h]

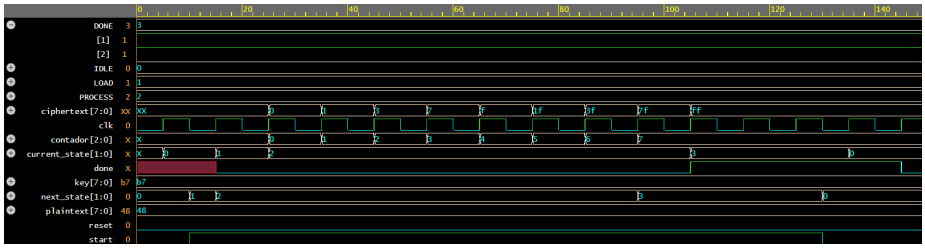

Figure 4: Diagrama de tempo do primeiro teste

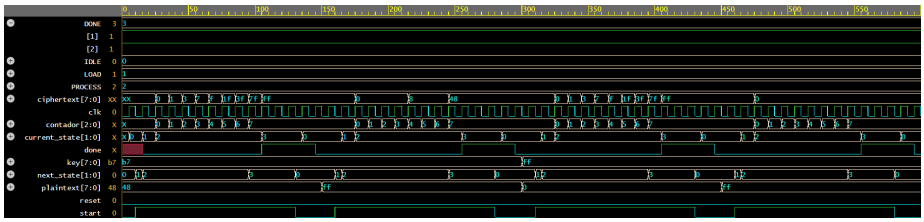

Figure 5: Diagrama de tempo de TODOS os testes

5 BÔNUS

Para cumprir com os requisitos bônus, foi instruído que uma palavra plaintext fosse cifrado por meio de uma key de tamanho menor. Para resolver esse problema, basta utilizar módulo assim que o contador ultrapassar o tamanho da key, fazendo com que a key "recomece".

```
if (current_state == PROCESS) begin
  ciphertext[contador] <= plaintext[contador] ^ key[contador % (tamanho_key)];

  if (contador < (tamanho_palavra-1)) begin
        contador <= contador + 1;
```

Figure 6: Indexando a posição dos bits usando módulo

Outro problema evidente que notei enquanto fui implementar essa funcionalidade, foi que a diferença no tamanho das palavras não pode ser alterada durante o programa e têm que ser definida de acordo com cada teste e alterada em seu respectivo design. Mas tentei fazer o código de uma forma em que basta trocar

simplesmente o tamanho das palavras(plaintext, key e ciphertext) na declaração delas e modificar o teste para que se adeque ao tamanho da palavra. Isso implica também na mudança de tamanho do contador.

```
module cifra_xor #(
  parameter tamanho_key = 8,
  parameter tamanho_palavra = 16
)
```

Figure 7: Novos tamanhos de palavra

Uma coisa interessante a respeito dessa versão é que ela funciona com a chave do mesmo tamanho que o plaintext também, ou seja, é um modelo de código que cumpre com a ideia base do projeto e com a parte bônus.

Por final, outra coisa que tive que modificar foi a quantidade de ciclos enviados ao circuito por teste, porque no circuito base, como eram somente 8 bits, eu usei 11 ciclos, para garantir que todas as operações ocorressem normalmente ($11 >= 8$). A adaptação que eu fiz foi estabelecer 19 ciclos, pois $19 >= 16$, assim, o circuito funciona normalmente e consegue passar por todos os bits.

```
),
        #(tempo_clk);
          start      = 1'b1;
        #(tempo_clk * 19);
```

Figure 8: Mudança nos ciclos

No geral, essas foram as mudanças que eu realizei, a lógica para mudança/permanência de estados continuou muito similiar com o projeto base, as alterações realizadas foram para adequar o circuito com o tamanho novo de palavras.

Como não podemos simplesmente usar o operador XOR como fizemos no projeto base, por conta do tamanho diferente entre palavra e chave, podemos tentar reverter a criptografia usando nosso próprio circuito. Fiz os testes e de fato, está funcionando de acordo.
