Calculadora em Verilog

SISTEMAS DIGITAIS - 98G02-04
Professor: Anderson Domingues
Turma: 10
Bernardo Fraga, João Victor Terra, Pedro Oliveira, Raul Costa.

Imagens e Tabelas

Imagem 1: esquemático do circuito.
Imagem 2: resultado das *waves* do *test bench*.
Imagem 3: display de sete segmentos representando o número "3".
Imagem 4: tabela com valores de dígitos e seu código em hexadecimal demonstrados da *wave*.

Objetivo

O objetivo desse projeto é criar uma calculadora básica, utilizando *Verilog*, capaz de realizar operações aritméticas básicas entre dois números - até um total de 32 bits -, sendo elas: adição, subtração e multiplicação (utilizando somas consecutivas) e o comando de igualdade, responsável por executar as operações. Também será possível limpar a calculadora e visualizar uma mensagem de erro quando o display chegar no limite de 32 bits.

Queremos fazer um arquivo de *testbench*, chamado tb_final.sv, para que seja possível testar o funcionamento da calculadora. Nele, iremos simular possíveis entradas de um usuário, testando situações incomuns e de possíveis erros. Utilizaremos a ferramenta *Questa* para simular o *clock* e o *reset*, assim como visualizar as ondas (sinais) subindo e descendo.

Metodologia

Antes de começar a desenvolver toda a lógica da calculadora, desenvolvemos dois arquivos responsáveis por controlar 8 displays de 7 segmentos cadas, os quais irão ser utilizados para mostrar os números da calculadora. São eles o Display.sv, responsável por mostrar um dígito em um display individualmente, e o Display_Ctrl.sv, responsável por usar o Display.sv para controlar as posições de cada dígito entre os 8 displays. Adicionamos também um arquivo sim.do para a execução do código e um wave.do para a geração da forma de onda.

O módulo principal do projeto é a Calculadora_Top.sv, ela é responsável por receber 3 informações: CMD, CLOCK, RESET. O CMD é o **comando** do usuário, de 4 bits, ele é levado para o módulo Calculadora.sv. Nesse módulo é onde ocorre toda lógica da calculadora.

Primeiramente, o código dentro do módulo Calculadora.sv utiliza um bloco sequencial "*always*", sensível à borda do ***clock*** e do ***reset***, implementado uma lógica simples com um bloco condicional dentro, o qual espera um dígito de 0 à 9 ou uma operação.

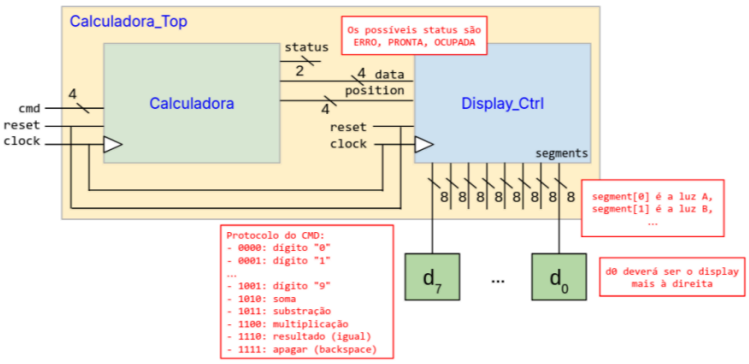

Imagem 1: esquemático do circuito.

Protocolo do CMD

- 0000: 0
- 0001: 1
- 0010: 2
- 0011: 3
- 0100: 4
- 0101: 5
- 0110: 6
- 0111: 7
- 1000: 8
- 1001: 9
- 1010: operação de soma
- 1011: operação de subtração
- 1100: multiplicação
- 1101: - (não utilizado)
- 1110: resultado
- 1111: apagar

Se identificar que é um dígito, ele atribui a um registrador chamado ***Reg1*** o valor desse dígito e continua atribuindo mais valores caso o usuário digite mais números. Se o programa identificar que não é um dígito, ele entende que só pode ser uma operação.

Se essa operação **não for** "igual a", o comando do usuário é armazenado como operador e a variável ***set_op*** recebe "1". Agora como é ***set_op*** é diferente de zero, o primeiro *else if* é ativo, fazendo com que o comando do usuário seja armazenado no ***Reg2***. Após isso, o programa, novamente, espera outra operação. Se essa operação for "igual a", então o programa entra no bloco condicional *"else if (cmd == 4'b1110)*", onde existe um case para identificar qual operação realizar.

Conforme o valor da operação, salvo anteriormente, o programa realiza quatro possíveis ações:

- **Adição:** Faz uma soma simples entre **Reg1** e **Reg2**.
- **Subtração:** Faz uma subtração simples entre **Reg1** e **Reg2**.
- **Multiplicação:** Usa somas em sequência para realizar a multiplicação.
    - Se Reg1 ou Reg2 for zero, a saída será zero.
    - Se não, o estado da calculadora passa para OCUPADO. Nesse estado, a cada ciclo do *clock* o valor de Reg1 é somado à saída e o contador incrementado até chegar ao valor de Reg2. Quando o contador for igual ao Reg2, o estado volta para PRONTA.

- **Backspace:** Apaga o último dígito inserido no **Reg1**, fazendo uma divisão por 10. Caso o valor de ***set_op*** já tenha mudado para 1, ele faz essa divisão no **Reg2**.

Para a lógica das posições no display, fizemos com que a cada dígito recebido, o dígito vai para o *dig* do display e a *pos* é incrementada em um, pulando assim para o próximo *display*.

Testagem e Conclusão

Para testar o código, fizemos um simples *test bench* o qual simula entradas no comando a cada ciclo de um *clock.* Encontramos erros nos testes, possivelmente erros de lógica na calculadora.sv. Percebemos que o primeiro *display* é pulado, mesmo utilizando uma lógica para que o *pos* não seja incrementado na primeira vez - o que foi resolvido inicializando pos com -1 -. Também tivemos problemas com números com mais de um algarismo, mesmo utilizando a lógica que multiplica o reg1/reg2 por 10 e em seguida acrescenta o comando do usuário. Nas testagens foi possível observar que os valores dos registradores e dos resultados só são mostrados dois clocks após a entrada simulada. Outro pequeno problema foi que, o último resultado fica se repetindo até acabar os displays, caso não recebe mais nenhum comando após ele. Acreditamos que este problema seja causado pelo envio do sinal de saída a cada ciclo de clock, assim como o problema da falta de operações com números acima de 9 seja causada por um erro na lógica do envio das posições do módulo calculadora para o módulo display_ctrl, através do sinal “pos”.

Test Bench utilizado

```
//Geração de clock
  initial clock = 0;
  always #5 clock = ~clock;


  initial begin
    // Inicialização
        cmd   = 4'b0000;
```

```
      reset = 1;
    #10 reset = 0;


    // ======================
    // Teste 1: 3 + 1 =
    // ======================
    #8  cmd = 4'b0011;  // 3
    #10 cmd = 4'b1010;  //+
    #10 cmd = 4'b0001;  //1
    #10 cmd = 4'b1110;  // =
    #20 reset = 1;
    #10 reset = 0;

    // ======================
    // Teste 2: 3 - 1 =
    // ======================
    #10 cmd = 4'b0011;  // 3
    #10 cmd = 4'b1011;  //-
    #10 cmd = 4'b0001;  //1
    #10 cmd = 4'b1110;  // =


    // ======= ERRO =========
    // Teste 3: 3 + 8 =
    // ======================
    #10  cmd = 4'b0011;  // 3
    #10 cmd = 4'b1010;  //+
    #10 cmd = 4'b1000;  //8
    #10 cmd = 4'b1110;  // =
    #20 reset = 1;
    #10 reset = 0;
    #100;

    $stop;
  end
```

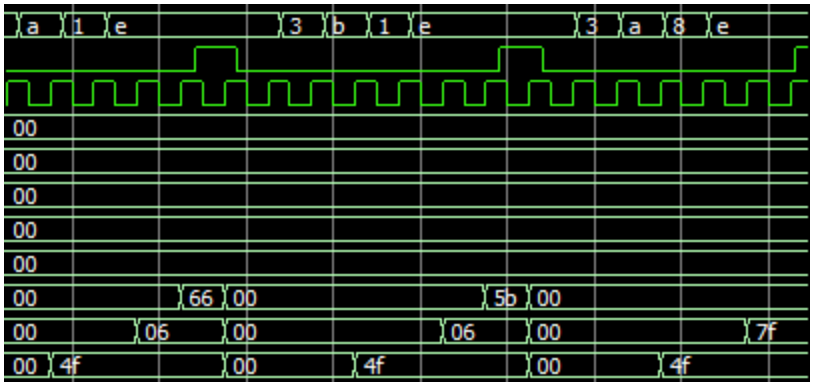

Imagem 2: resultado das *waves* do *test bench*

Os números acima são representações em hexadecimal dos dígitos que aparecem nos displays, representados em binário, por isso fizemos a tabela abaixo para ajudar na compreensão:

Ex.: Para mostrar o dígito 3, é necessário que os segmentos a,b,c,d,g estejam ativos, formando o código binário “1111001”, que em Hexadecimal é representado por 0x4f

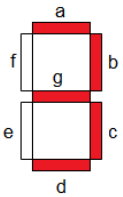

Imagem 3: *display* de sete segmentos representando o número “3”

Tabela De Compreensão Dos Valores Na *Wave*

Dígito/Op	Hexa
0	3F
1	06
2	5B
3	4F
4	66
5	6D
6	7D

7	07
8	7F
9	6F
Soma	a
Subt	b
Mult	c
Vazio	d
Igual	e

Imagem 4: tabela com valores de dígitos e seu código em hexadecimal demonstrados da *wave*.

Referências

1. ELECTRONICS FUN. 7 Segment HEX Decoder. Disponível em: https://electronics-fun.com/7-segment-hex-decoder/

2. INTEL. Intel® Quartus® Prime Design Software - Questa Edition. Disponível em: https://www.intel.com.br/content/www/br/pt/software/programmable/quartus-prime/questa-edition.html

3. ASIC-WORLD. Verilog Syntax. Disponível em: https://www.asic-world.com/verilog/syntax.html

4. FPGA TUTORIAL. How to Write a Basic Verilog Testbench. Disponível em: https://fpgatutorial.com/how-to-write-a-basic-verilog-testbench/

5. GITHUB. GitHub. Disponível em: https://github.com/