Projeto Somador/Subtrator

Guilherme Altmeyer Soares;
Igor Correa Domingues de Almeida;

Universidade Estadual do Oeste do Paraná- Unioeste

Aritmética Binária

Antes de entrarmos nos pormenores dos sistemas que envolvem operações aritméticas, que seja apresentado como foi pensada as operações fundamentais (adição e subtração) para números binários.

Adição Binária:

As adições envolvendo números binários se baseiam nas seguintes regras:

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 (e “vai 1” para o dígito de ordem superior - carry)
1 + 1 + 1 = 1 (e “vai 1” para o dígito de ordem superior - carry)

Carry: A palavra carry, em inglês, é aquele número que a gente "sobe" quando a operação de dois números ultrapassa o limite daquela representação.

Complemento de dois:

Na parte tangente a representação dos números binários surgiram algumas ideias de como separar os números positivos e negativos. A que mais se destacou devido a sua fácil implementação e entendimento por parte da máquina foi o complemento de dois.

Esse tipo de representação utiliza o bit mais significativo para determinar se o número será positivo ou negativo. Para um número positivo, seu valor é representado na forma binária direta, sendo acrescido um bit de sinal 0 na frente do MSB do número

binário. Para um número negativo, o valor é representado na forma de complemento a 2 e um bit de sinal 1 é colocado na frente do MSB.

O complemento a 2 é realizado da seguinte forma:

- Invertemos todos os bits do número binário, ou seja, tudo que é 0 vira 1 e vice-versa.
- Somamos um (1) ao valor invertido.

Subtração Binária:

Conceitualmente na matemática não existe subtração, tudo oque temos é a soma do complemento do número. Claro que existem formas de subtrair números binários sem recorrer a esse método, porém, quando o nosso objetivo passa ser a implantação, essa forma de pensar nos ajudará bastante.

Então para esse trabalho, quando quisermos subtrair um número um número, trataremos ele com o complemento de dois anteriormente explicado e utilizaremos a adição binária.

Overflow:

Por definição overflow ocorre quando o valor calculado não pode ser representado no espaço destinado à ele. No nosso projeto final Overflow só irá ser representado (se ocorrer) na saída do nosso último full adder, caso que irei detalhar mais ao longo do projeto.

Circuito Aritmético

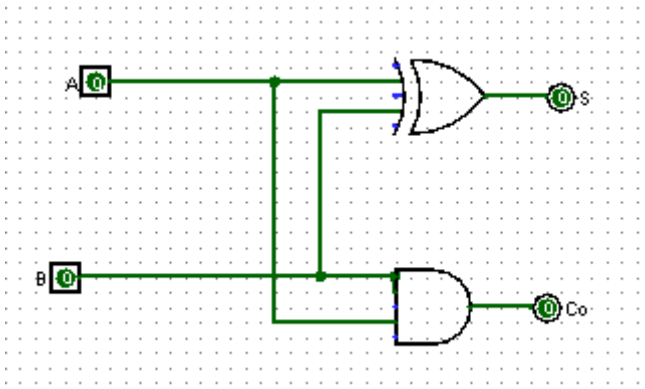
Half Adder:

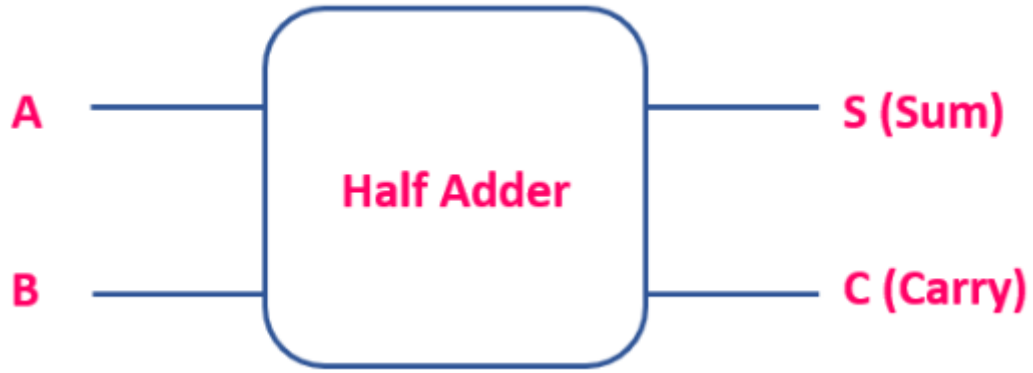
Com base no conhecimento adquirido nas operações binárias vamos montar uma tabela verdade que soma 2 números binários (a e b):

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Representando cada número por 1 bit, podemos então montar um circuito que possui como entradas: A e B. E como saída, a soma dos algarismos representada por S e o respectivo transporte de saída C (carry).

Extraindo as expressões da tabela verdade (álgebra de boole) percebemos que a saída **S** é dada por: **A xor B.** E que o carry **C** é representado por: **A.B**





Full Adder:

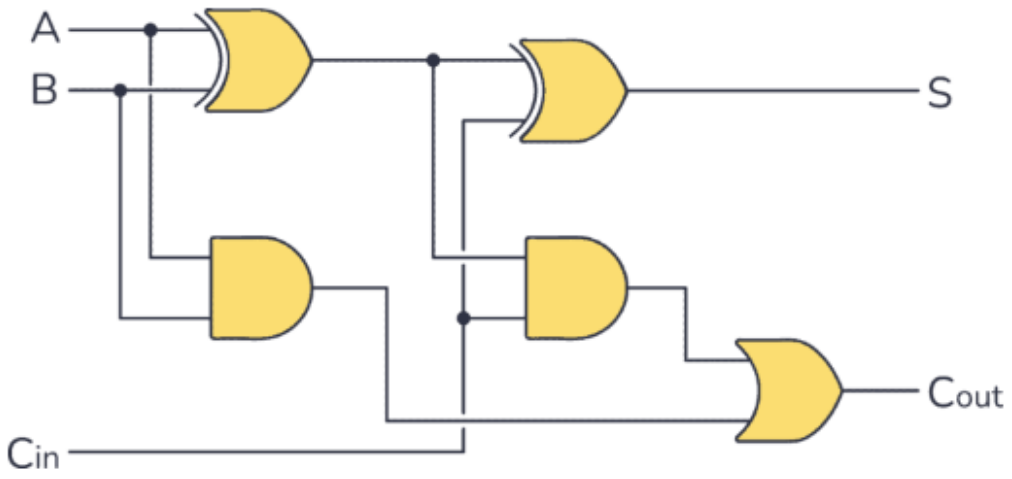
O meio somador possibilita efetuar a soma de números binários com 1 algarismo. Porém, quando se deseja utilizar mais de 1 algarismo, o circuito torna-se insuficiente pois não possibilita a introdução do transporte de entrada.

Então para somar números binários com mais de 1 algarismo, basta somarmos a coluna, levando em conta o transporte de entrada, que é o C da coluna do half adder.

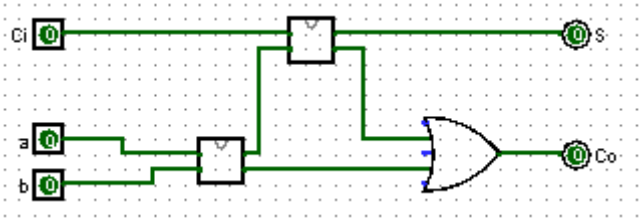
A	B	Cin	S	Cout
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Por Álgebra de Boole e processo de simplificação obtemos:
S: (A xor B) xor Cin
Cout: B.Cin + A.Cin + A.B

Construindo o circuito dessa expressão obtemos:



Porém como podemos observar a partir do circuito, é que o Full Adder se utiliza de 2 Half Adder para sua implementação sendo assim:

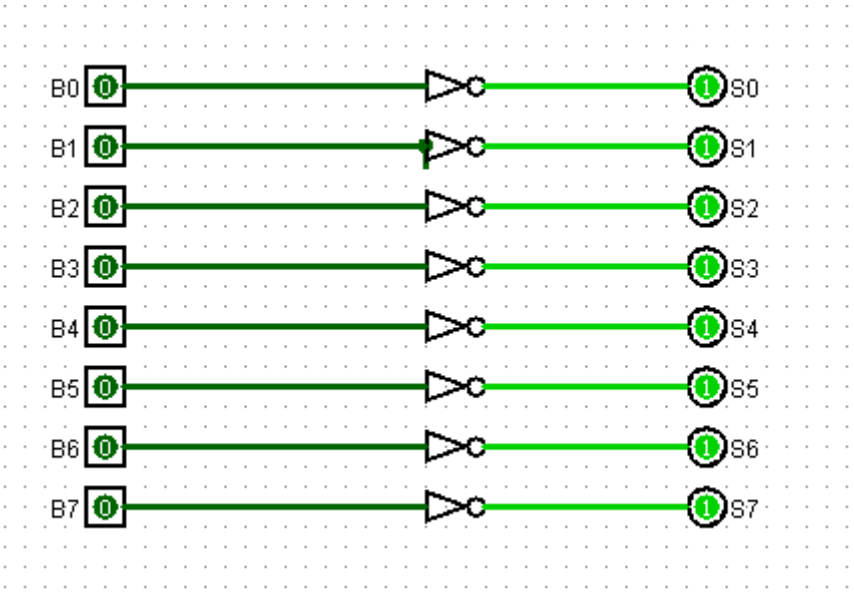


Full Suber:

Como detalhado anteriormente, nossa operação binária de subtração será efetuada pela soma do complemento do subtraendo.

Tomamos como (B) a entrada que será realizada a ação de complemento ou não, Basta aplicarmos (B) e (negação de B) a um mux que escolherá quais bits receber de acordo com a operação desejada.

Começaremos desenvolvendo um circuito que negue B, este é bem simples:



Agora para montarmos um multiplex de 2 entradas e 8 bits iremos utilizar 8 mux de 2x1:

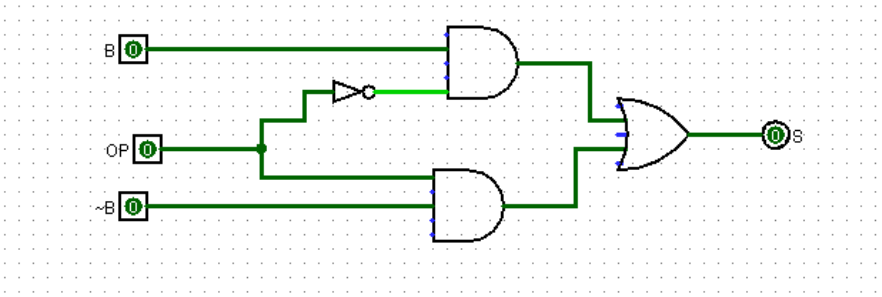
T.V.	$Canal_0$	$Canal_1$	SELECT	Z
Sit 0	0	0	0	0
Sit 1	0	0	1	0
Sit 2	0	1	0	0
Sit 3	0	1	1	1
Sit 4	1	0	0	1
Sit 5	1	0	1	0
Sit 6	1	1	0	1
Sit 7	1	1	1	1

Extraindo o circuito e simplificando por VDK:

DVK	$\overline{Canal_1}$		$Canal_1$	
$\overline{Canal_0}$	0	0	1	0
$Canal_0$	1	0	1	1
	$\overline{SELECT}$	SELECT		$\overline{SELECT}$

Por fim temos:
S: (Canal 0.~select) + (Canal1.Select).

A circuito montado fica assim:



Utilizando 8 mux de 2 x 1 estabelecemos um de 2x8 que irá realizar o complemento de 2 em B quando for necessário.

Latência:

A latência em um sistema digital se refere ao atraso que ocorre entre a entrada de um sinal ou evento no sistema e a saída correspondente. Ela é a medida do tempo que um sistema leva para processar uma determinada entrada e produzir uma resposta. A latência é uma consideração crítica em muitos sistemas digitais, especialmente em aplicações em tempo real, onde a resposta rápida é essencial.

Em nosso projeto foi requisitado um segundo caso onde a latência entre as portas lógicas tem um valor de 4ns.

Circuito Somador/Subtrator

Problema:

Construir um circuito que faça a soma e subtração de números até 8 bits e que nos retorne a resposta e se deu ou não overflow.

Entradas:

O circuito deve possuir 3 entradas: uma entrada (op) de 1 bit indicando se a operação realizada será adição ou subtração;

2 entradas (A e B) de 8 bits cada representando os números a serem efetuadas as operações.

Saídas:

O circuito deve possuir 3 saídas: uma saída (S) de 8 bits, com a resposta da nossa operação;

Uma saída (CarryOut) de 1 bit, indicando o bit que sai da operação;

E uma saída (Overflow) de 1 bit indicando casos em que ocorre overflow ou não.

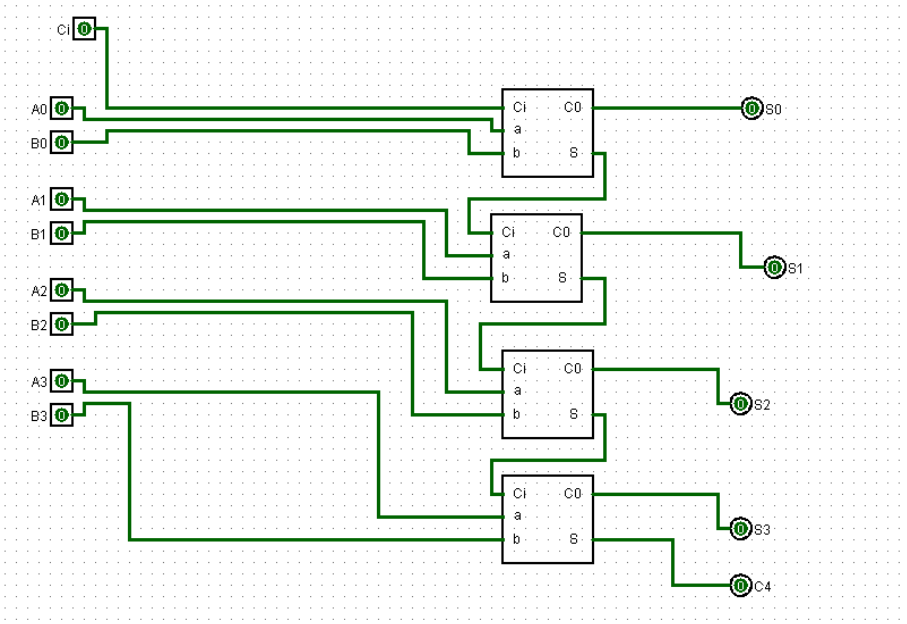
. Para desenvolver esse circuito se torna necessário quebrar o problema em partes menores e no final juntar tudo, para isso o projeto foi desenvolvido da seguinte forma:

- Construção de um circuito somador de 8 bits.
- Construção de um circuito not de 8 bits
- Construção de um circuito mux2x8.
- Construção de um circuito overflow de 1 bit;

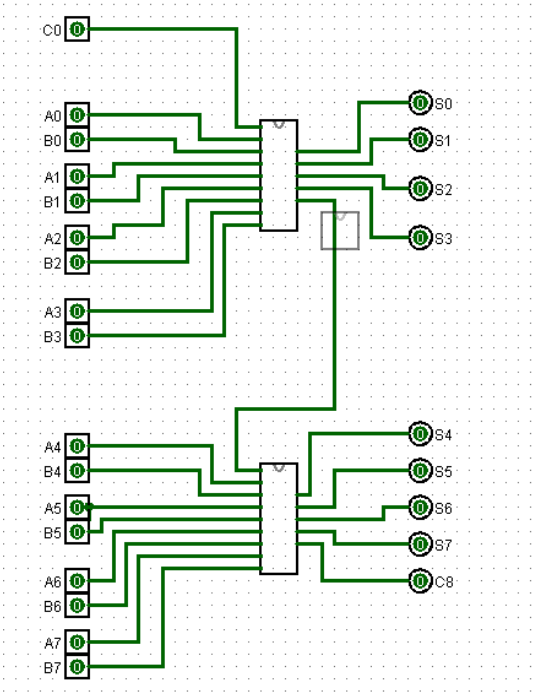
Adder 8 bits:

Como já foi explicado a lógica por trás do circuito adder, neste bloco irei apenas tratar da implementação deste circuito.

O circuito adder de 4 bits é feito a partir de 4 full adders:



E agora para realizarmos um de 8 bits, basta usarmos 2 adders de 4 bits:



A porta C que entra tem o mesmo valor lógico da porta operação, e segue incrementando as seguintes adições com ‘1’ ou com ‘0’ seguindo as regras da adição em binário.

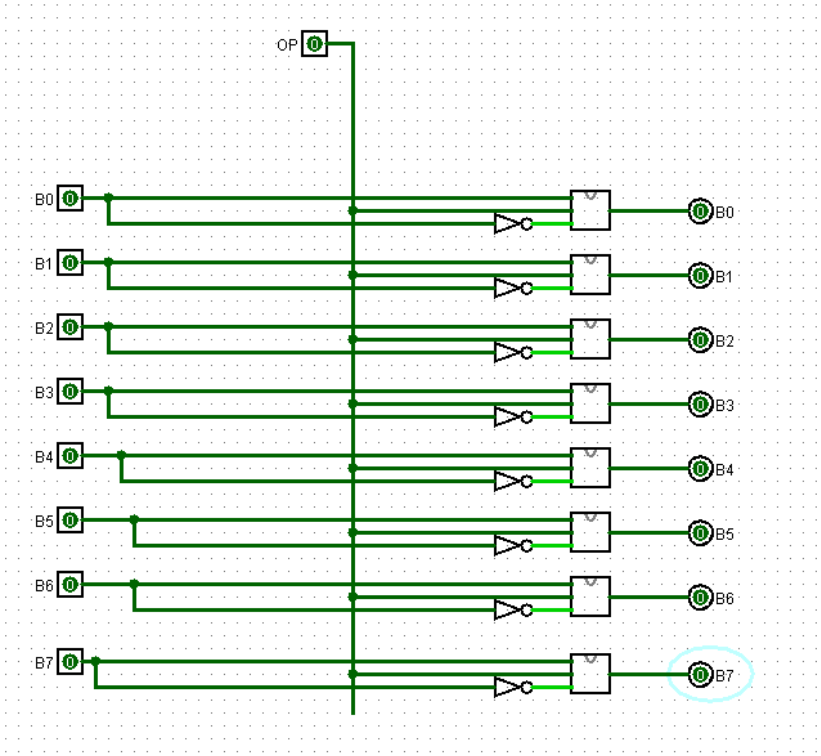
Complemento de 2

Como foi demonstrado no bloco full subber, a operação de subtração será baseada na adição do complemento de B.

Um circuito mux de 2x8 será responsável pela aplicação do complemento de 2 quando necessário.

Por definição, um multiplexador é um circuito combinacional com N entradas e 1 saída, controlado por N2 sinais de controle. De acordo com o padrão de valores aplicados nos sinais de controle, uma das entradas é copiada para a saída.

Para nossa implementação o circuito mux ficou assim:



A saída B será redirecionada diretamente no nosso full adder de 8 bits anteriormente explicado.

Overflow:

No circuito é necessário relatar as situações em que ocorre o overflow, no nosso caso, overflow ocorre com números maiores que 127 e menores que -128. O circuito Overflow recebe 3 entradas.

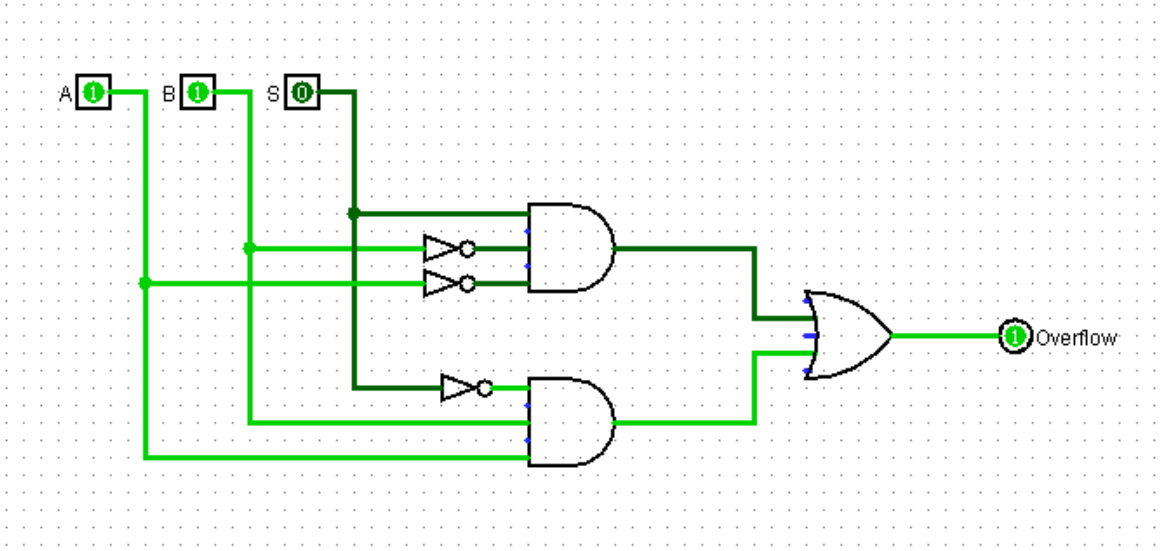
Importante destacar que só receberemos o overflow no último bit da adição entre a e b.

O circuito é montado a partir de 3 entradas: A, B, S;

A situação overflow só é verdadeira (ocorre) quando a entrada A e B são iguais e diferente de S, correspondendo assim que o número extrapolou a representação

a	b	s	$((\neg a \wedge \neg b) \wedge s) \vee ((a \wedge b) \wedge \neg s)$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	0
0	1	0	0
0	0	1	1
0	0	0	0

Construindo o circuito lógico equivalente temos:



Agora que já abordamos todos os pormenores do nosso circuito vamos pra implementação.

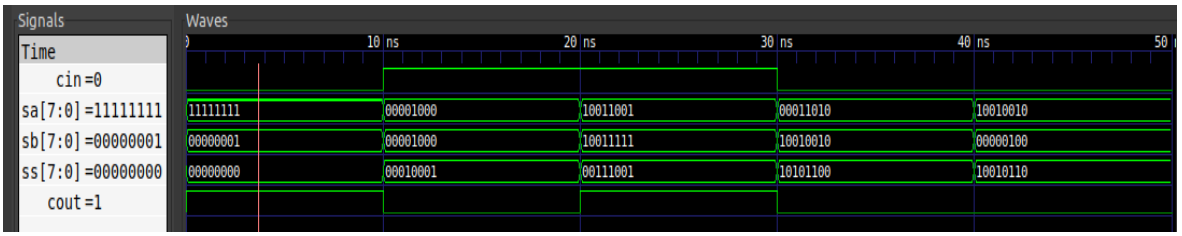
SIMULAÇÃO FULLADDER

Situação 01:

Os casos testados para a simulação são os seguintes:

```
    u_tb : process
    begin
            sa <= "11111111";
            sb <= "00000001";
            scin <= '0';
            wait for 10 ns;

            sa <= "00001000";
            sb <= "00001000";
            scin <= '1';
            wait for 10 ns;

            sa <= "10011001";
            sb <= "10011111";
            scin <= '1';
            wait for 10 ns;

            sa <= "00011010";
            sb <= "10010010";
            scin <= '0';
            wait for 10 ns;

            sa <= "10010010";
            sb <= "00000100";
            scin <= '0';
            wait for 10 ns;
        wait;
    end process;
end architecture;
```

Na simulação em forma de onda os resultados foram os seguintes:



No primeiro teste, é feita a soma com os números “11111111” (255) e “00000001” (1) de entrada, e com cin = ‘0’, as saídas foram ss = “00000000” e cout = ‘1’, pois a soma extrapola os 8 bits, tendo como resultado “100000000” (256).

No segundo, são somados “00001000” (8) com “00001000” (8), com cin = ‘1’, e as saídas são “00010001” (17), com cout = ‘0’.

No terceiro, é somado “10011001” (153) com “10011111” (159), com cin = ‘1’, a saída ss é igual a “00111001” (57) e o cout é igual a ‘1’, totalizando “100111001” (313).

No quarto, a entrada a = “00011010” (26), a entrada b = “10010010” (146) e a entrada cin = ‘0’, a saida s = “10101100” (172) e o cout = ‘0’.

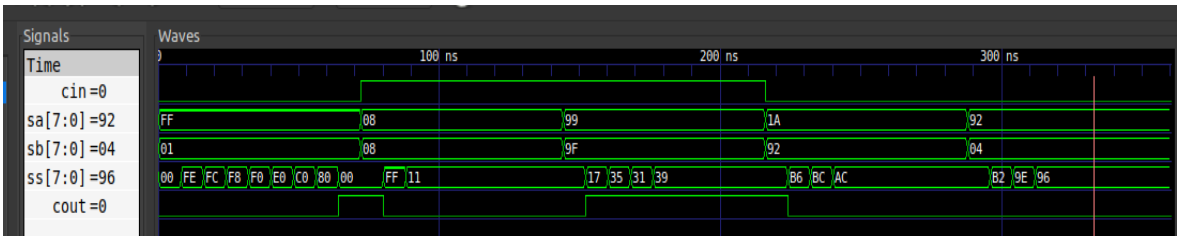
No último, entram “10010010” (146) em a, “00000100” (4) em b e ‘0’ em cin, sai “10010110” (150) em s e ‘0’ em cout.

Situação 02:

Os casos testados são os mesmos:

```
u_tb : process
begin
        sa <= "11111111";
        sb <= "00000001";
        scin <= '0';
        wait for 72 ns;

        sa <= "00001000";
        sb <= "00001000";
        scin <= '1';
        wait for 72 ns;

        sa <= "10011001";
        sb <= "10011111";
        scin <= '1';
        wait for 72 ns;

        sa <= "00011010";
        sb <= "10010010";
        scin <= '0';
        wait for 72 ns;

        sa <= "10010010";
        sb <= "00000100";
        scin <= '0';
        wait for 72 ns;
    wait;
```

E os resultados também, mas devido a adição de latência no código, há a necessidade de aumentar o tempo de espera após cada teste e também há impacto na representação em ondas, como mostrado na imagem:



Nota-se a diferença no "ss", o número final vai se formando aos poucos, relativo a latência de cada adder de 1 bit, como são 8 bits, o resultado são 8 * 2 * 4 ns (Porta XOR e porta OR) e mais 8 ns para visualização.

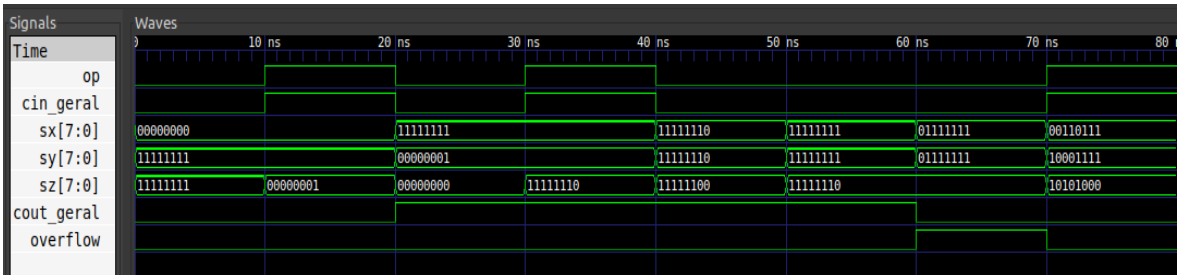
SIMULAÇÃO SIGNEDADDER

Situação 01:

Os casos testados para a simulação são os seguintes:

```
begin

    sx <= "00000000";   -- '0'
    sy <= "11111111";   -- "-1"
    op <= '0';
    Cin_Geral <= '0';   -- Recebe o mesmo que a operação
    wait for 10 ns;     --sz <= "11111111" (-1) scout <= '0' soverf <= '0'

    op <= '1';
    Cin_Geral <= '1';
    wait for 10 ns;     --sz <= "00000001" (1) scout <= '0' soverf <= '0'

    sx <= "11111111";   --"-1"
    sy <= "00000001";   -- '1'
    op <= '0';
    Cin_Geral <= '0';
    wait for 10 ns;     --sz <= "00000000" (0) scout <= '1' soverf <= '0'

    op <= '1';
    Cin_Geral <= '1';
    wait for 10 ns;     --sz <= "11111110" (-2) scout <= '1' soverf <= '0'

    sx <= "11111110";   -- "-2"
    sy <= "11111110";   -- "-2"
    op <= '0';
    Cin_Geral <= '0';
    wait for 10 ns;     --sz <= "11111100" (-4) scout <= '1' soverf <= '0'

    sx <= "11111111";   -- "-1"
    sy <= "11111111";   -- "-1"
    Cin_Geral <= '0';
    wait for 10 ns;     --sz <= "11111110" (-2) scout <= '1' soverf <= '0'

    sx <= "01111111";   -- "127"
    sy <= "01111111";   -- "127"
    Cin_Geral <= '0';
    wait for 10 ns;     -- sz <= "11111110" ("-2") scout <= '0' soverf <= '1'
                        -- Resposta esperada "254", mas ocorreu o overflow

    sx <= "00110111";   -- "55"
    sy <= "10001111";   -- "-113"
    op <= '1';
    Cin_Geral <= '1';
    wait for 10 ns;     -- sz <= "10101000" (-88) scout <= '0' soverf <= '0'

wait;
```

Na simulação do GTKWave ficou assim:



É possível ver que todos os testes tiveram o resultado esperado apontado nos comentários do VHDL, por exemplo:

O primeiro teste era a soma (op = ‘0’) de 0 (“00000000”) e -1 (“11111111”), com cin = ‘0’, cujo resultado esperado é -1 (“11111111”) e com as outras saídas sendo 0 (overflow e cout). O segundo era a subtração (op = ‘1’) de 0 e -1, com cin = ‘1’, cujo resultado foi 1 (“00000001”), também sem cout e overflow (Lembrando que o cin é igual a operação e está relacionado a conversão para negativo em complemento de 2, onde se invertem todos os dígitos e se adiciona 1).

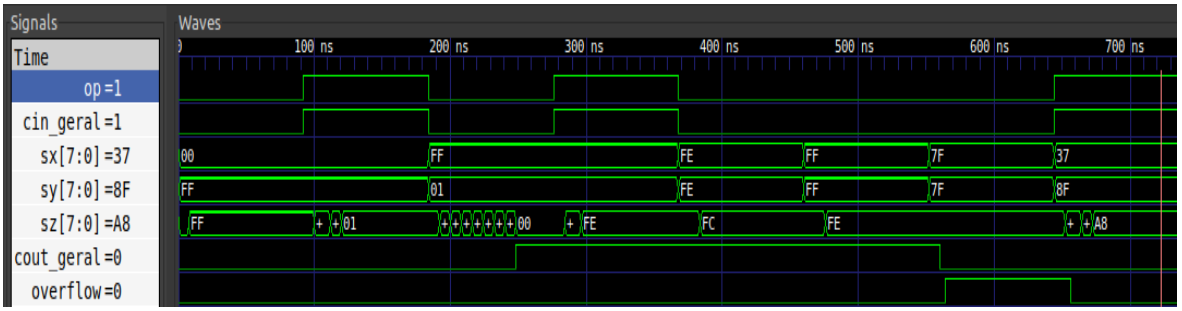
No sétimo teste, onde há a soma entre 127 (“01111111”) e 127 e a saída esperada é 254, mas como isso é maior do que o somador consegue representar, a saída final ficou -2 (“11111110”) e a Overflow Flag recebeu ‘1’.

Situação 02:

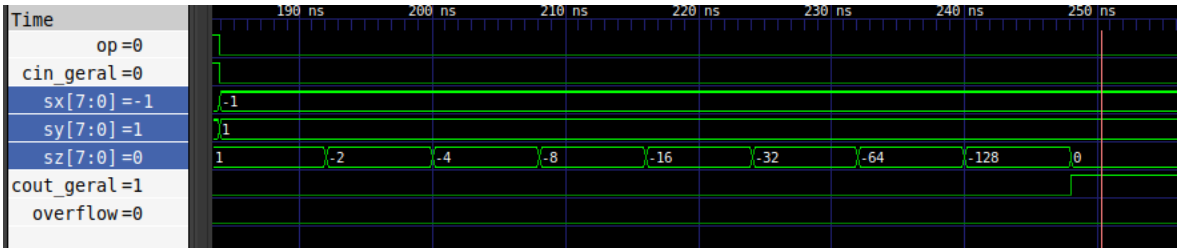
Os casos testados são os mesmos:

```
    u_tb : process
    begin

        sx <= "00000000";   -- '0'
        sy <= "11111111";   -- "-1"
        op <= '0';
        Cin_geral <= '0';   --Cin_geral recebe o mesmo que operação
        wait for 92 ns;     --sz <= "11111111" (-1) scout <= '0' soverf <= '0'
             -- 64(Fadder8) + 12(Mux2x8) + 4(not) + 12(overflow)

        op <= '1';
        Cin_geral <= '1';
        wait for 92 ns;     --sz <= "00000001" (1) scout <= '0' soverf <= '0'

        sx <= "11111111";   --"-1"
        sy <= "00000001";   -- '1'
        op <= '0';
        Cin_geral <= '0';
        wait for 92 ns;     --sz <= "00000000" (0) scout <= '1' soverf <= '0'

        op <= '1';
        Cin_geral <= '1';
        wait for 92 ns;     --sz <= "11111110" (-2) scout <= '1' soverf <= '0'

        sx <= "11111110";   -- "-2"
        sy <= "11111110";   -- "-2"
        op <= '0';
        Cin_geral <= '0';
        wait for 92 ns;     --sz <= "11111100" (-4) scout <= '1' soverf <= '0'

        sx <= "11111111";   -- "-1"
        sy <= "11111111";   -- "-1"

        wait for 92 ns;     --sz <= "11111110" (-2) scout <= '1' soverf <= '0'

        sx <= "01111111";   -- "127"
        sy <= "01111111";   -- "127"
        wait for 92 ns;     -- sz <= "11111110" ("-2") scout <= '0' soverf <= '1'
                            -- Resposta esperada "254", mas ocorreu o overflow
        sx <= "00110111";   -- "55"
        sy <= "10001111";   -- "-113"
        op <= '1';
        Cin_geral <= '1';
        wait for 92 ns;     -- sz <= "10101000" (-88) scout <= '0' soverf <= '0'

    wait;
    end process;
end architecture;
```

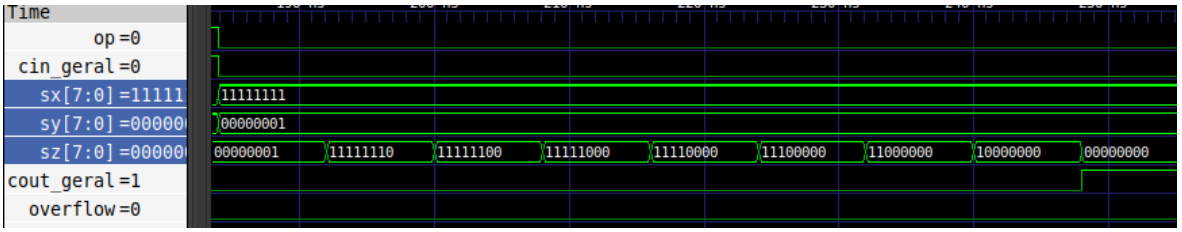
Com diferença na latência, com 64 ns dos FullAdder, 12 para o Multiplexador, 4 para o inversor e 12 para o Overflow, totalizando 92 ns, fazendo a simulação do GTKWave ficar assim:



Os resultados são os mesmos da Simulação 01, mas é possível perceber, devido a latência, o tempo que demora para cada saída ser gerada, o terceiro teste é o que melhor permite ver isso:



O resultado começa em 1, vai para -2, então -4, -8, -16, -32, -64, -128 e finalmente chega em 0, que é o resultado final esperado. Esse teste deixa bem visível a evolução na quantidade de bits de cada operação, indo de 1 a 8 bits, como se espera de um RippleAdder:



Referências:

(1) Alexandre Santos de la Vega: Apostila de Teoria para Circuitos Digitais;
(2) António Joaquim Esteves João Miguel Fernandes: Apontamentos Teóricos da Disciplina de SISTEMAS DIGITAIS 1 ;
(3) Idoeta e Capuano - Elementos de Eletrônica Digital;
(4) Slides da Aula;