UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

LUANE STEFFANE LOPES

PEDRO SADER AZEVEDO

PROJETOS 1 E 2

Projetos realizados para a disciplina Circuitos Lógicos (EA772),
da Faculdade de Engenharia Elétrica e de Computação (FEEC),
da Universidade Estadual de Campinas (UNICAMP),
como avaliação parcial.

Professor: Dr. José W. M. Bassani

CAMPINAS - SP

2022

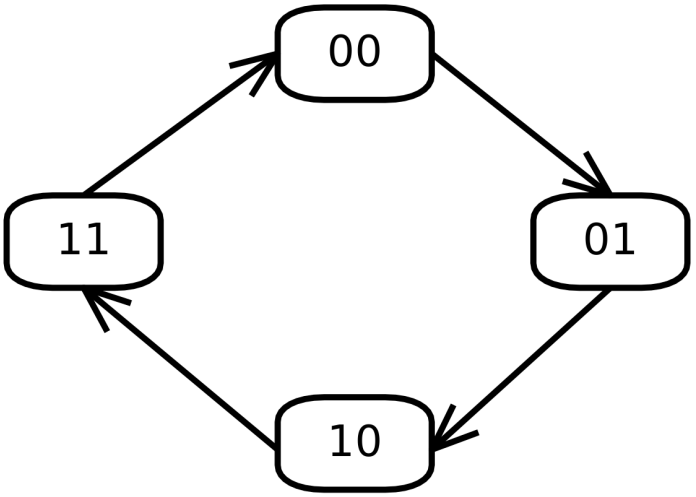
PROJETO 01 - Memória Somente Leitura

Este projeto tem como objetivo a montagem de uma unidade de Memória Somente Leitura (*Read Only Memory*, ROM). Esse circuito lógico permite a seleção de diferentes blocos de memória, com cada um deles contendo um programa de quatro instruções de quatro bits. Assim que o programa é escolhido por meio de um *push button*, as instruções são lidas sequencialmente a cada pulso de um *clock*.

Para isso, foram necessários diversos componentes, cujos projetos detalharemos a seguir:

- **Contador**

A função do contador no circuito foi estabelecer a sequência dos blocos e das linhas. Como temos quatro blocos e quatro linhas, são precisos dois bits para representar todos os estados do contador, representados no diagrama abaixo:



Optamos inicialmente por projetar um contador assíncrono, usando flip flops JK. Para os próximos passos, precisaremos da tabela de excitação desse tipo de flip flop, a qual podemos construir facilmente a partir da sua tabela verdade:

J	K	Intuição	Z^{n+1}
0	0	mantém	Z^n
0	1	*reset*	0
1	0	*set*	1
1	1	inverte	$\overline{Z^n}$

Z^n	Z^{n+1}	Intuição	Dedução	J	K
0	0	mantém ou re*set*	$(J = 0 \bullet K = 0) + (J = 0 \bullet K = 1) \equiv (K = 1 + K = 0) \bullet (J = 0) \equiv (J = 0)$	0	x
0	1	inverte ou *set*	$(J = 1 \bullet K = 1) + (J = 1 \bullet K = 0) \equiv (K = 1 + K = 0) \bullet (J = 1) \equiv (J = 1)$	1	x
1	0	inverte ou *reset*	$(J = 1 \bullet K = 1) + (J = 0 \bullet K = 1) \equiv (J = 1 + J = 0) \bullet (K = 1) \equiv (K = 1)$	x	1
1	1	mantém ou *set*	$(J = 0 \bullet K = 0) + (J = 1 \bullet K = 0) \equiv (J = 1 + J = 0) \bullet (K = 0) \equiv (K = 0)$	x	0

Assim, a partir da sequência de estados ilustrada acima e da conhecida tabela de excitação do flip flop JK, construímos uma tabela relacionando os dígitos Z_1 e Z_0 às entradas dos flip flops:

Z_1^n	Z_0^n	J_1	K_1	Z_1^n
0	0	0	x	0
0	1	x	0	1
1	0	1	x	1
1	1	x	1	0

Se assumimos $x = 0$, temos: $J_1 = \overline{Z_1} \bullet Z_0$, $K_1 = Z_1 \bullet Z_0$

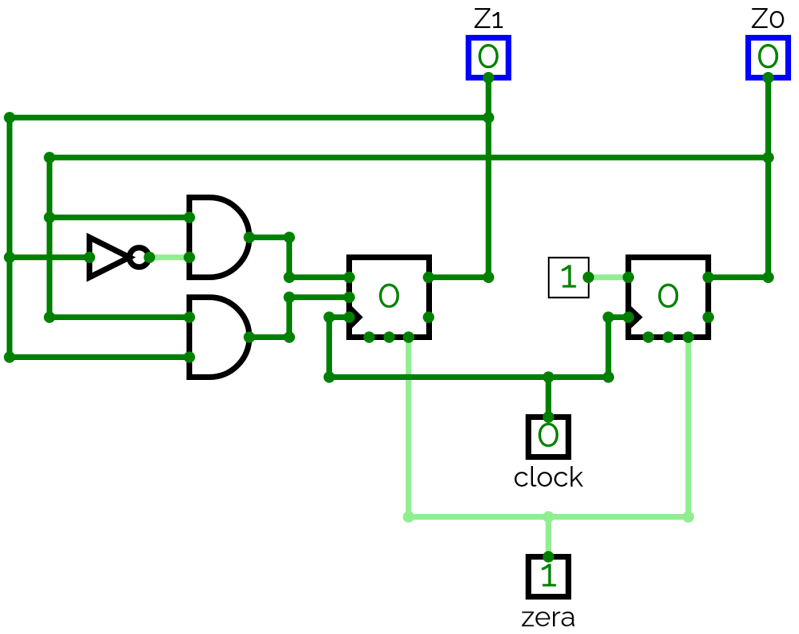
Z_1^n	Z_0^n	J_0	K_0	Z_0^n
0	0	1	x	1
0	1	x	1	0
1	0	1	x	1
1	1	x	1	0

Se assumimos $x = 1$, temos: $J_0 = 1$, $K_0 = 1$.

Como as duas entradas do flip flop JK são idênticas, podemos usar um flip flop T (que é simplesmente um flip flop JK com o mesmo valor de input T em suas duas entradas). Assim, a tabela do dígito menos significativo do contador fica:

Z_1^n	Z_0^n	T	Z_0^n
0	0	1	1
0	1	1	0
1	0	1	1
1	1	1	0

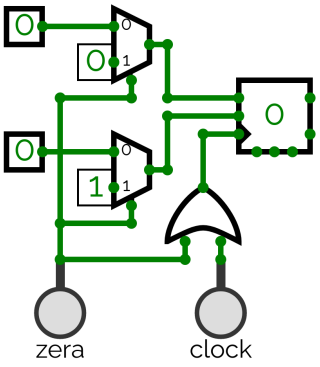
Com isso, já podemos construir o circuito do contador:



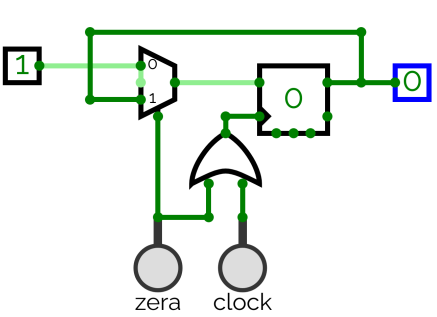
Esse contador é incrementado a cada pulso de *clock* (ou pressionamento de *push button*). Além disso, incluímos um input que zera o contador, cuja utilidade será começar a leitura de instruções desde a primeira (instrução 00) quando um novo programa for selecionado.

Usamos a funcionalidade de zerar incluída nos próprios flip flops fornecidos pelo software de simulação, para deixar o circuito mais enxuto e fácil de compreender. No entanto, como essa não pode ser considerada uma "funcionalidade básica" dos flip flops, decidimos projetá-la também.

Se lembramos que *J* e *K* são análogos à *set* e *reset*, fica claro que devemos atribuir o valor de 0 ao primeiro e o valor de 1 ao segundo. Assim, para zerar o dígito mais significativo do contador, basta selecionar esses valores usando multiplexadores.



Agora, lembramos que T é a abreviação de *toggle*, que significa "alternar" em inglês. Assim, para zerar o dígito menos significativo, precisamos de $T = 1$ (para alternar) se o valor atual do flip flop for 1, e $T = 0$ (para manter) se o valor atual do flip flop já for 0. Por isso, basta selecionar o próprio valor do flip flop para zerá-lo.



- **Decodificador**

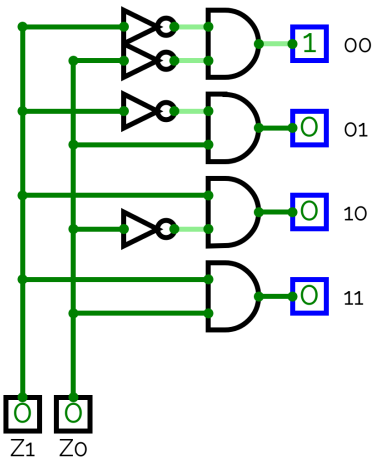
Para selecionar as linhas e blocos a partir do número armazenado nos contadores, fez-se necessário um decodificador. Esse componente produz output nulo para todas as suas saídas, exceto aquela que corresponde ao número representado pelos bits de entrada, como mostra a tabela abaixo:

Z_1^n	Z_0^n	V	W	X	Y
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Cada saída tem apenas uma configuração de dígitos que a seleciona, então podemos usar os mintermos de cada uma delas para elaborar suas expressões lógicas:

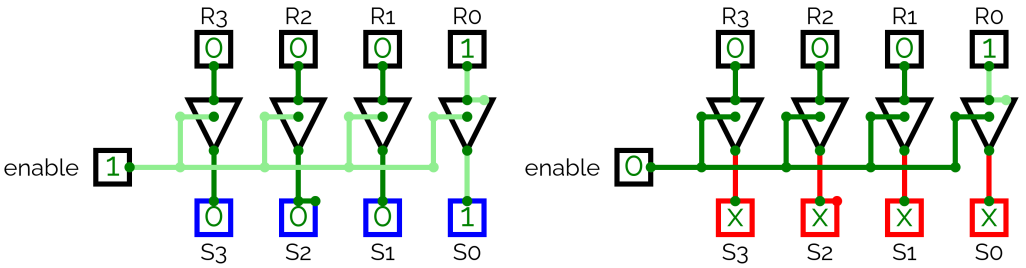
$V = \overline{Z_1} \bullet \overline{Z_0}, W = \overline{Z_1} \bullet Z_0, X = Z_1 \bullet \overline{Z_0}, Y = Z_1 \bullet Z_0.$

A partir disso, construímos o circuito do decodificador:



- **Linha (ou Instrução)**

As instruções do programa são lidas sequencialmente, no mesmo barramento de quatro bits. Isso significa que elas devem se conectar ao barramento de forma mutuamente exclusiva, o que foi feito com um banco de *tri-state buffers* ligados à mesma entrada (*"enable"*):

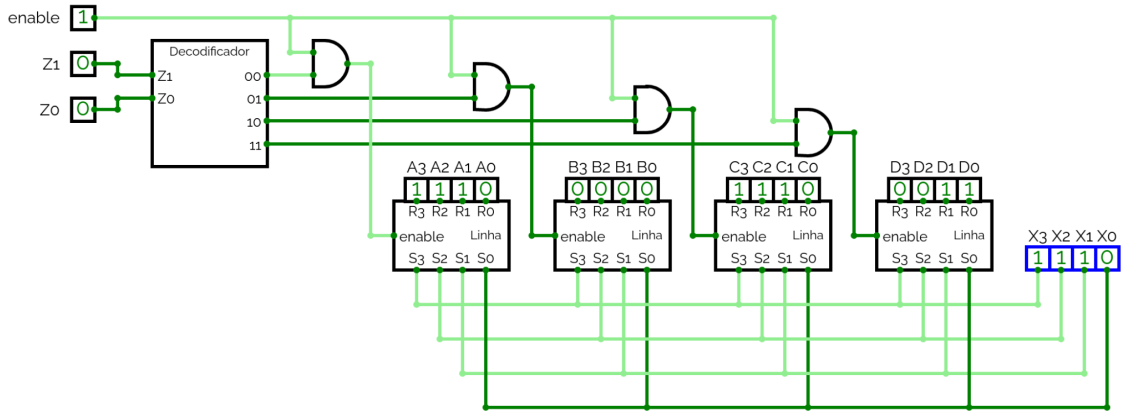


Para melhorar a clareza do circuito, esse componente será representado daqui em diante por uma caixa preta, com legenda "Linha".

- **Bloco (ou Programa)**

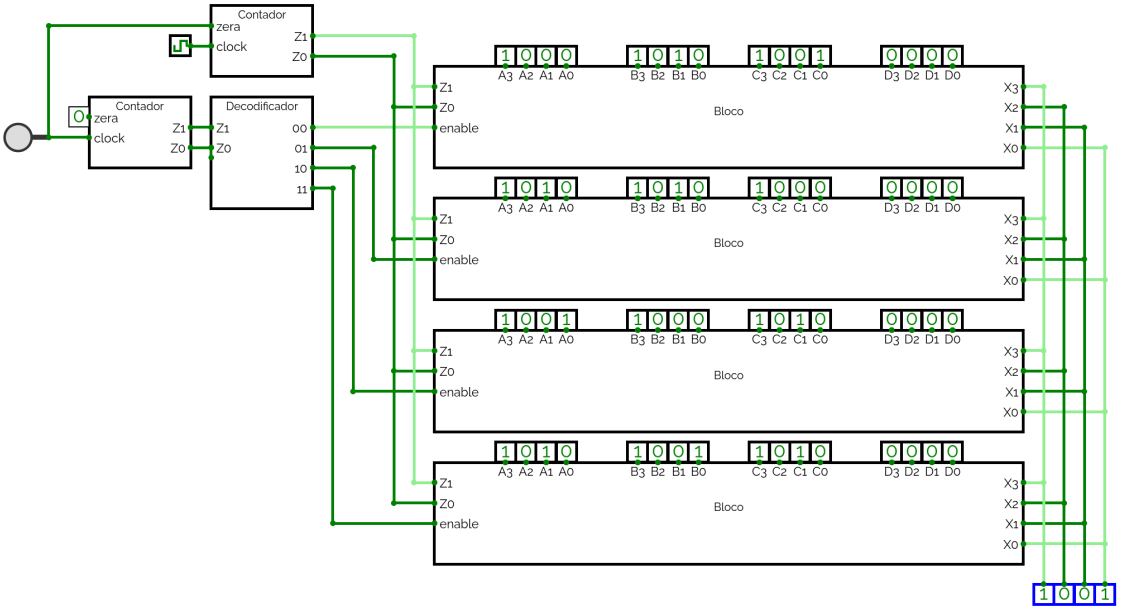
O último componente a projetar antes de montar a Memória Somente Leitura, é o componente de bloco de instruções (ou programa). Esse componente tem uma entrada *"enable"*, assim como o componente anterior, além de dois dígitos Z_1 e Z_0.

Os dígitos Z são encaminhados para um decodificador, cujas saídas estão ligadas a portas lógicas AND juntamente com o *"enable"* do próprio bloco. Isso garante que apenas linhas do bloco selecionado sejam lidas.



Agora que projetamos todos os componentes necessários para a Memória Somente Leitura, podemos enfim montá-la. Para isso, conectamos um *push button* à entrada "*clock*" de um contador, cujos dígitos de saída são encaminhados para um

decodificador. As saídas do decodificador são ligadas às portas *"enable"* de cada bloco do circuito, completando a seleção do programa por meio do *push button*.



Outro contador, dessa vez com a porta "*clock"* de fato ligada a um *clock*, fica responsável pela leitura sequencial das linhas/instruções. Sempre que a escolha de programa muda, ao pressionar o botão, o contador que seleciona as linhas é zerado, para iniciar a leitura do programa a partir de sua primeira instrução. Uma simulação do circuito completo pode ser conferida no CircuitVerse.

Projeto 02 - Unidade Lógica Aritmética

Em primeiro lugar, objetiva-se com este projeto realizar a construção de uma Unidade Lógica Aritmética (ULA). Além do mais, para conseguirmos concluir esse objetivo, será necessário algumas etapas e componentes que integrarão nossa ULA. A princípio, tivemos que montar um contador de 4 bits com Carry Input, Carry Output, detecção de número negativo, Overflow e resultado da soma igual a zero. Além disso, montamos um circuito decodificador para o usuário determinar qual operação deveria ser realizada, dentre Adição, Subtração e Complemento.

- **Zera**

Para criar o módulo Zera, utilizamos nas entradas 4 bits, sendo eles: w0, w1, w2, w3, além disso temos z = 1, no qual a variável z está relacionada em instituir quando o nosso módulo será ativado e z = 0, que assegura quando não será ativado. Em relação às

saídas, temos 4 bits: y0, y1, y2, y3, isso consiste em y*i* = w*i*, quando o bloco estiver inativo e y*i* = 0 quando o bloco estiver ativo. Para um módulo zera de 4 bits, utilizamos 4 portas ANDs, visto que a partir da tabela verdade abaixo, obteve-se a função $\mathbf{yi} = \bar{z} \cdot$ w*i*.

Tabela Verdade: Nesta TV temos nossa saída y*i*.

z	**w*i***	**y*i***
0	0	0
0	1	1
1	0	0
1	1	0

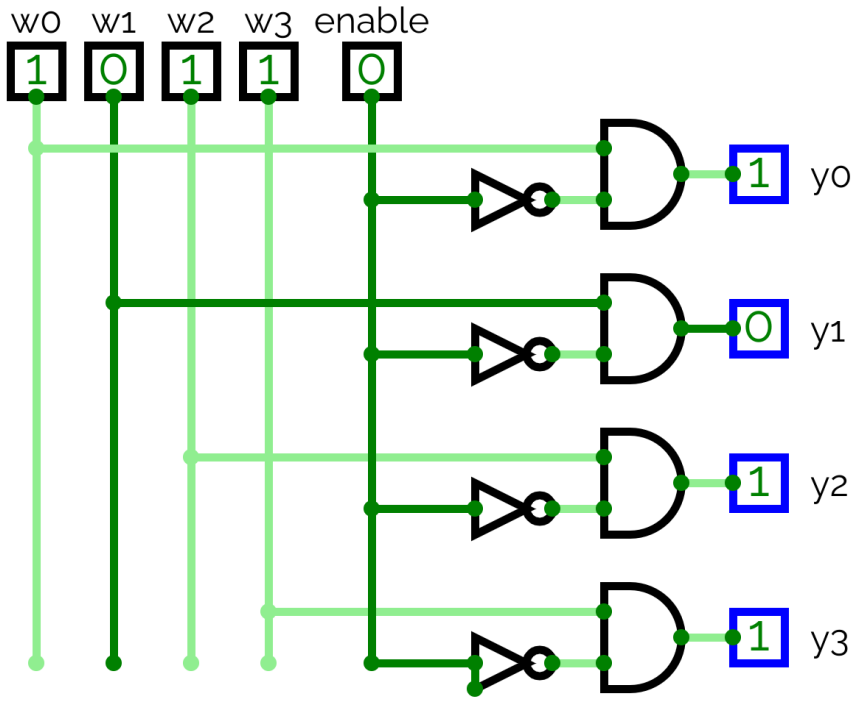


Imagem - Circuito Zera

- **Complementa**

A função do módulo complementa é inverter os bits de sua entrada, por exemplo, se acionarmos a sua função com uma das suas entradas igual à 1, a saída deverá ser 0. Para montarmos um complementa de um bit, sabe-se que precisamos de uma porta XOR de duas entradas, no qual uma serviria como a entrada do bit a ser

complementado, e a outra entrada para acionar a função. Tendo isso em vista, adaptamos o módulo complementa de um bit para 4 bits.

Nesse módulo, as entradas têm seus 4 bits sendo: l0, l1, l2 e l3, com uma variável N, na qual N = 0 determina que o bloco *não será ativado* e N = 1 determina que o bloco *será ativado*. Outrossim, nossas saídas por sua vez, serão: m0, m1, m2 e m3 - considerando que m*i* = l*i* quando N = 0 e que N = 1, quando for igual ao complemento de l*i* ($\overline{li}$).

Tabela Verdade

N	**l*i***	**m*i***
0	0	0
0	1	1
1	0	1
1	1	0

Obtivemos: $\overline{N}$l*i* + N$\overline{li}$, que representa-se como Ox = cmp $\oplus$ *i*x para as devidas saídas: m0, m1, m2 e m3.

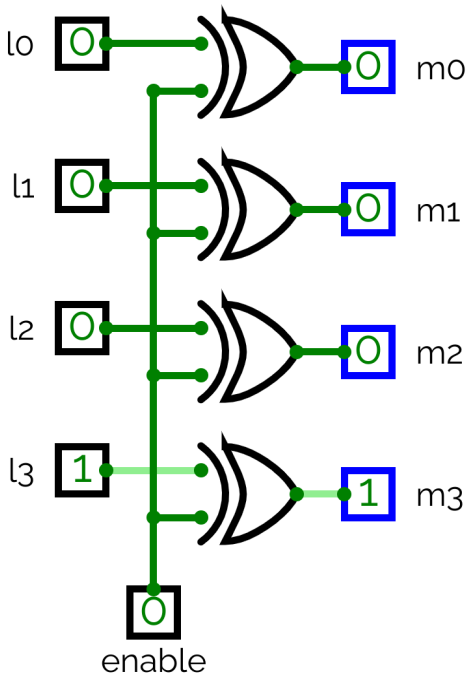


Imagem 2 - Circuito complementa de um número de 4 bits

- **Somador Completo de 4 bits**

Para implementação do somador de 1 bit completo, foram formadas a tabela da verdade e os mapas de Karnaugh para as saídas: S e C out.

X	Y	Ca In	Saída S	C out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Mapa de Karnaugh S

S/XY	0	1	11	10
0	0	1	0	1
1	1	0	1	0

Mapa de Karnaugh C out

S/XY	0	1	11	10
0	0	1	1	0
1	0	1	1	1

Sabe-se que um somador de 4 bits é integrado por 4 somadores de 1 bit associados. Sendo assim, sabendo como forma-se um somador completo de 1 bit, de entradas X e Y que se somam e um carry de entrada (Ca In), na qual as saídas são S (saída S) e carry de saída (C out), podemos construir um somador de 4 bits. A partir dos mapas acima, obteve-se as expressões: $S = Y \oplus X \oplus Cin$ e $C\ out = XY + XCin + YCin$.

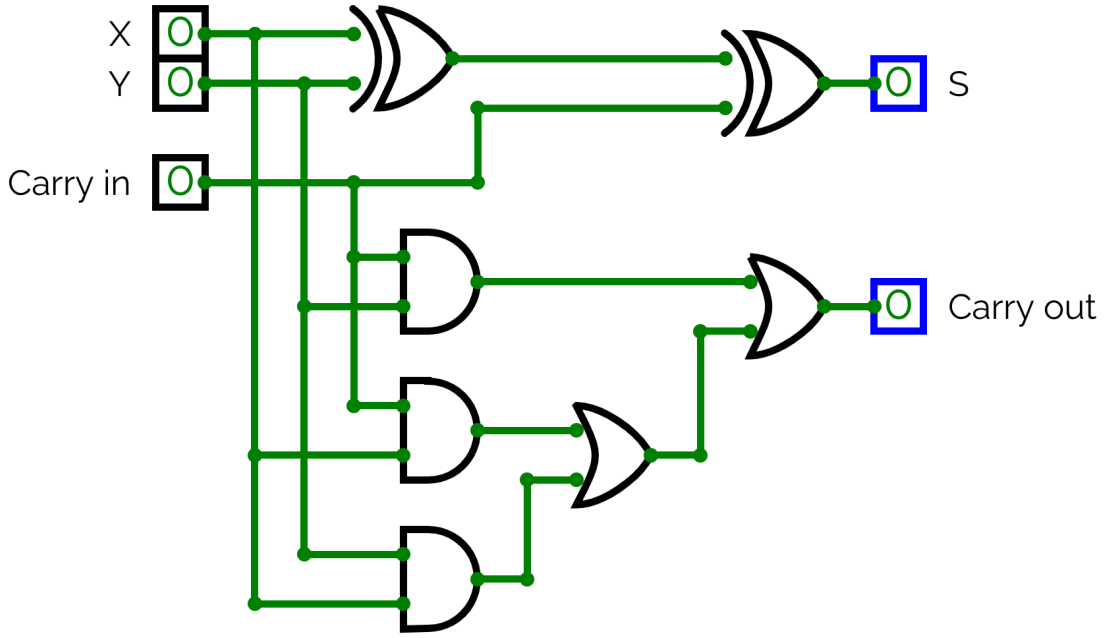


Imagem - Circuito Somador de 1 bit completo com Carry In e Carry Out

No somador de 4 bits, agregamos 4 barramentos, conectando os carrys. Além do mais, temos: Carry - nesse caso ela identifica quando houver carry do bit mais significativo para fora; Overflow - ocorre apenas quando o XOR entre o carry do bit mais significativo e o carry do número anterior for 1; Zera - vai detectar quando a saída for nula (todas as saídas precisam ser igual a 0) e Negativo - o número final apenas é negativo, a partir do momento em que ele for 1.

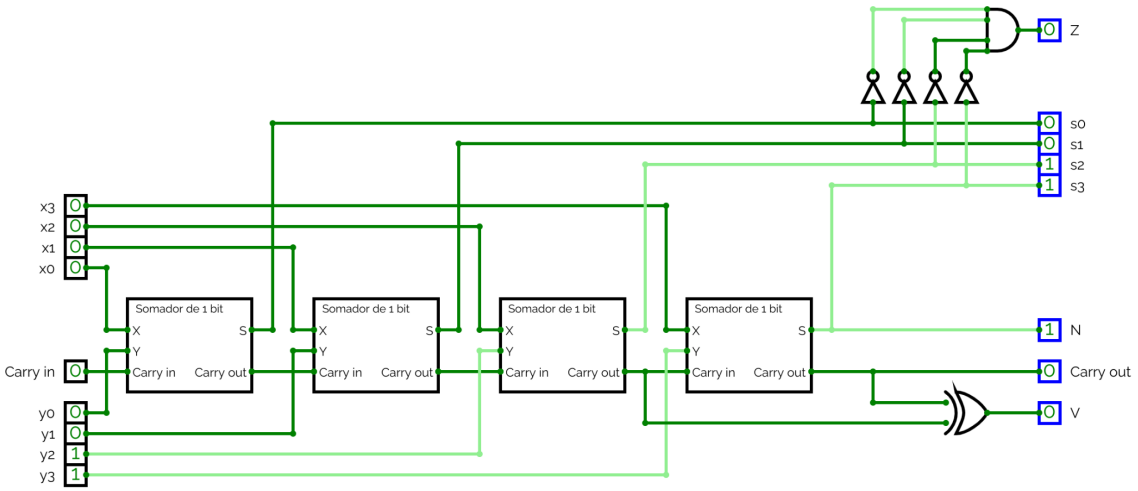


Imagem - Circuito Somador de 4 bits completo com Cin, Cout, Overflow, Zera e Negativo.

- **Decodificador**

Foi necessário montar um decodificador que verificasse a entrada do usuário e determinasse qual operação o circuito deveria realizar. Nota-se que o bloco *Complementa* soma 1 ao número, transformando a em (- a - 1) para que o número seja de fato complementado, devido aos números estarem representados em complemento de 2.

Tabela: Temos abaixo a tabela com as saídas Cx, Cy, Zy e o nosso Carry, as quais estão descritas suas possíveis combinações que representam as operações a serem realizadas na ULA. Observa-se que entramos com os números x e y, ainda que utilizamos o complemento de 2, isso porque "-a" se oponha em sinal e magnitude ao "a".

Cx	Cy	Zy	Carry	Resultado
0	0	0	0	x + y
0	0	0	1	x + y + 1
0	0	1	0	x
0	0	1	1	x + 1
0	1	0	0	x - y - 1
0	1	0	1	x - y - 1
0	1	1	0	x
0	1	1	1	x + 1
1	0	0	0	-x -1 + y
1	0	0	1	y - x
1	0	1	0	-x - 1
1	0	1	1	-x
1	1	0	0	-x - 1 - y - 1
1	1	0	1	-x - y - 1
1	1	1	0	-x -1 + y
1	1	1	1	-x

Tabela: Na tabela abaixo contém as combinações que satisfazem as operações de soma, subtração e complementação. Operações selecionadas a partir da entrada do usuário. Neste caso, nossas entradas são: p0, p1, p2.

p2	p1	p0	Cx	Cy	Zy	Carry	Resultado
0	0	0	0	0	0	0	Soma (x+y)
0	1	0	0	1	0	1	Subtração (x-y)
1	0	0	1	0	0	1	Subtração (y-x)
1	0	1	1	0	1	1	Complementação (-x)

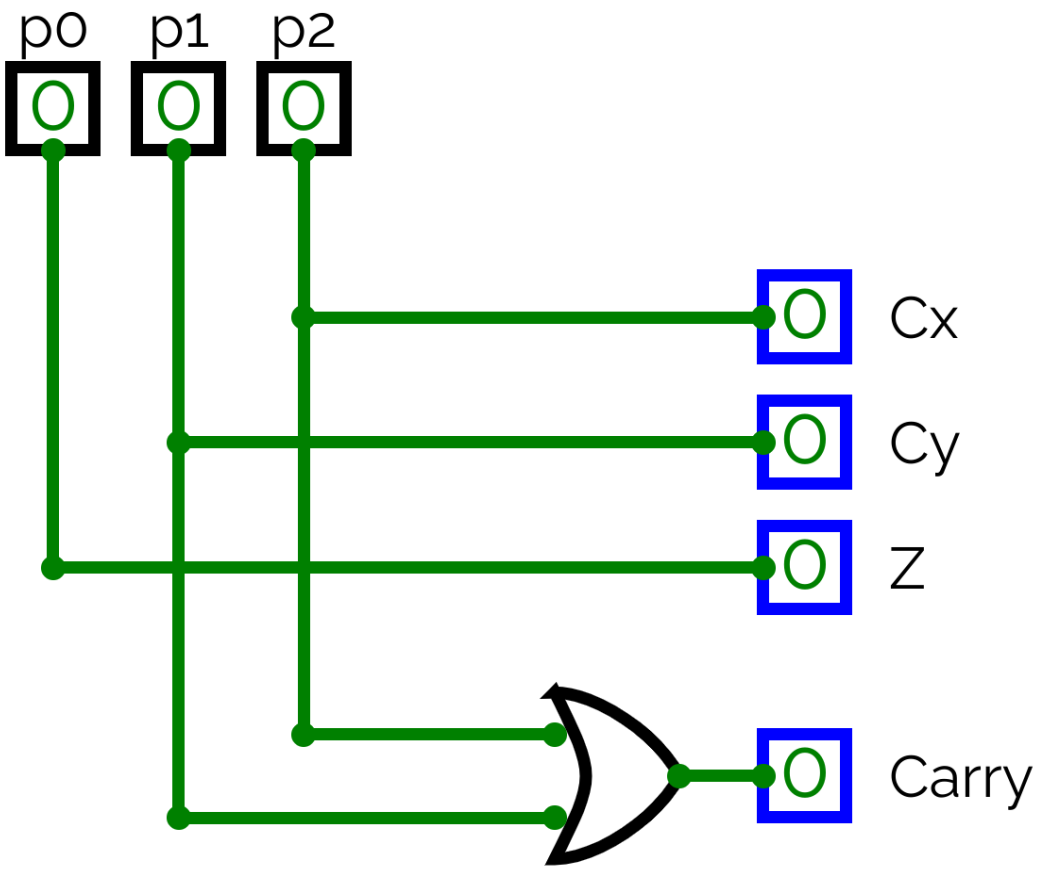


Imagem - Circuito Decodificador

- **ULA**

Para a implementação da ULA, utilizamos dois módulos do circuito Complementa, um módulo do circuito Zera, um módulo do Decodificador de operações e um módulo do somador completo de 4 bits com as funções de Zero, Carry, Overflow e Negativo. Com todos os módulos, criamos o circuito abaixo, cuja simulação pode ser conferida no CircuitVerse.

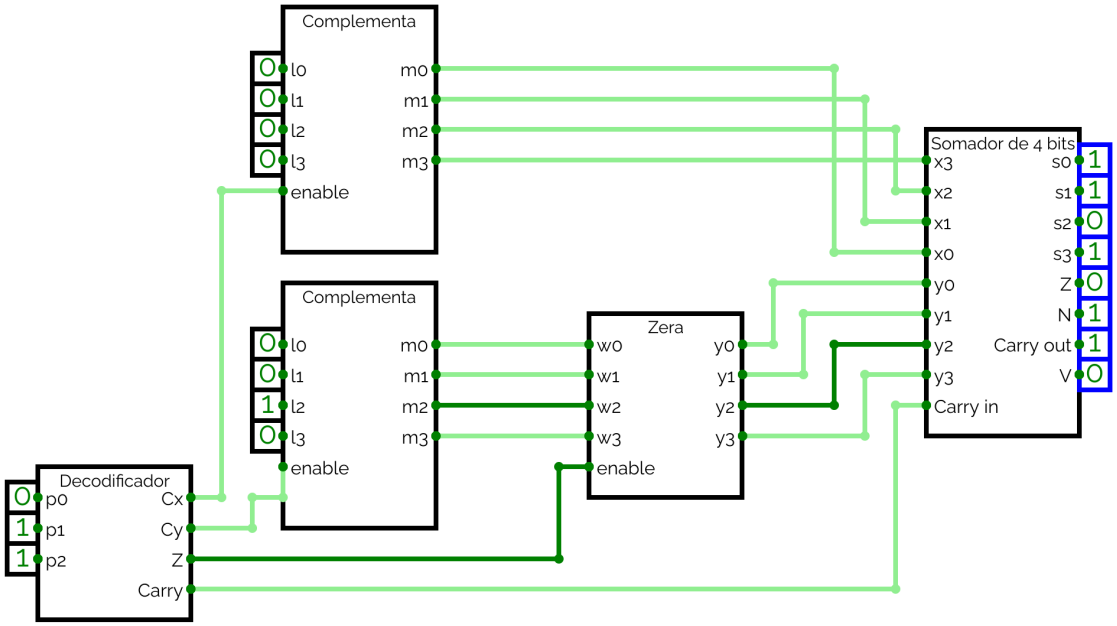


Imagem - Circuito ULA (simulação)

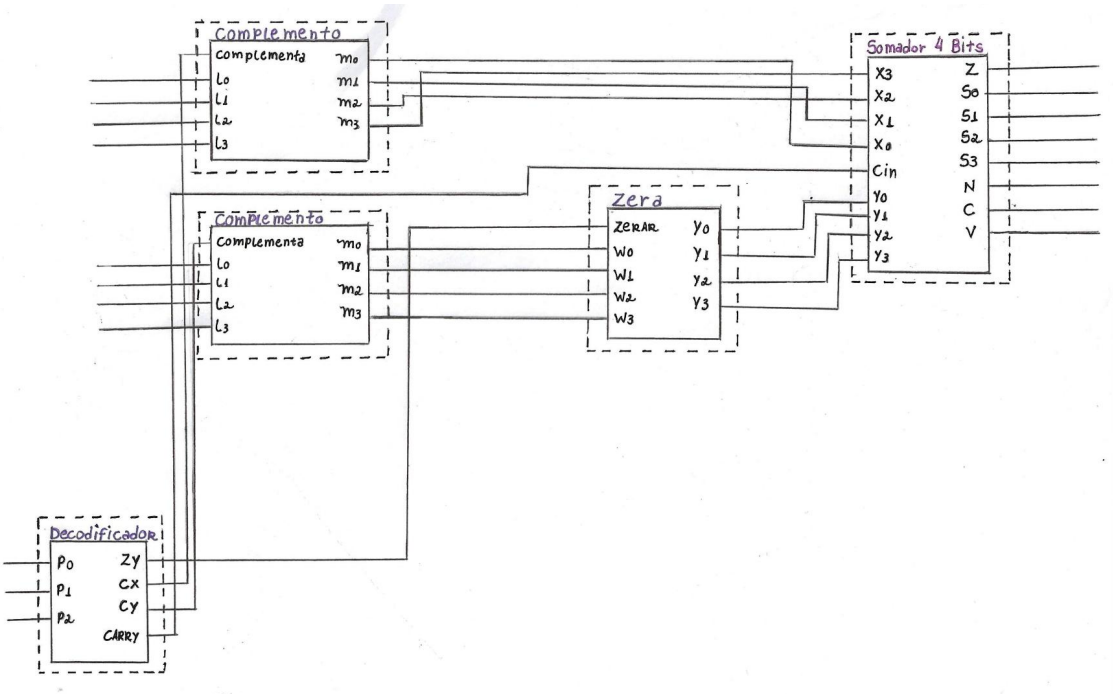


Imagem - Circuito ULA (desenho)