Henrique Coutinho Layber e Renan Moreira Gomes

IMPLEMENTAÇÃO DE UMA UNIDADE LÓGICA E ARITMÉTICA COM PORTAS LÓGICAS BÁSICAS

Professor Dr. João Paulo de Almeida

Universidade Federal do Espírito Santo

Ciência da Computação

Vitória, Espírito Santo, 2018

Henrique Coutinho Layber e Renan Moreira Gomes
Universidade Federal do Espírito Santo
Vitória 2018

Sumário

Conceituando a ALU

Unidade Lógica Aritmética (*Arithmetic Logic Unit – ALU*)é uma unidade lógica aritmética é um elemento central em uma CPU (Unidade de Processamento Central). Ela executa operações lógicas e aritméticas básicas diversas em dados de entrada, mas podemos juntá-las e usar das basicas para formar ALUs mais complexas. Iremos explicar passo a passo como projetar uma ALU que opere sobre dados de 8 bits, e como usar a mesma para montar uma ALU de 16 bits, como uma calculadora básica.

Especificação

Nossa ALU final irá conter as seguintes operações em cima das entradas A e B de 16 bits:

- `ADD` soma (`A + B`);
- `SUB` subtração (`A – B`);
- `XOR` bit-a-bit (`A XOR B`);
- `OR` bit-a-bit (`A OR B`);
- `AND` bit-a-bit (`A AND B`);
- `SHL` Shift left (`A << 1` desloca os bits de A para a esquerda);
- `SHR` Shift right (`A >> 1` desloca os bits de A para a direita).

Além dos operandos A e B de 8 bits, nossa ALU irá conter uma entrada OP de 3 bits que indica a operação a ser realizada com os operandos. A ALU irá seguir a seguinte tabela para a operação OP:

A ALU inicial também terá um `carry` de entrada (Cin), que permitirá depois concatenar duas ALUs de 8 bits para formar uma ALU de 16 bits, e as saídas de ambas deverão funcionar da seguinte forma:

Valor de OP	Saída
000	ADD Soma (A + B)
001	SUB subtracão (A − B)
010	XOR bit-a-bit (A XOR B)
011	OR bit-a-bit (A OR B)
100	AND bit-a-bit (A AND B)
101	SHL Shift left (A << 1)
110	SHR Shift right (A >> 1)
111	Não usada

Tabela a: Valor de OP e respectiva operações

- Uma saída C indicando `carry` (1 em C indica que houve `carry`) (no caso de subtração, indica `borrow`, no caso da operação `SHL`, o carry recebe o bit mais significativo sendo deslocado; no caso de `SHR`, recebe o bit menos significativo, em soma, indica Overflow);
- Uma saída Z que indica se todos os bits da saída são iguais a `0` (`1` em Z indica que todos os bits da saida S são `0`);
- Uma saída S de 8 bits com o resultado numérico da operação, e dois dígitos de display de 7 segmentos para mostrar o resultado.[1]

1 Não será feito display hexadecimal para a ALU de 16 bits, pois a sua implementação é mais complexa.

Observe na imagem abaixo, todas as entradas e saídas que serão utilizadas como base para a definição da ALU:

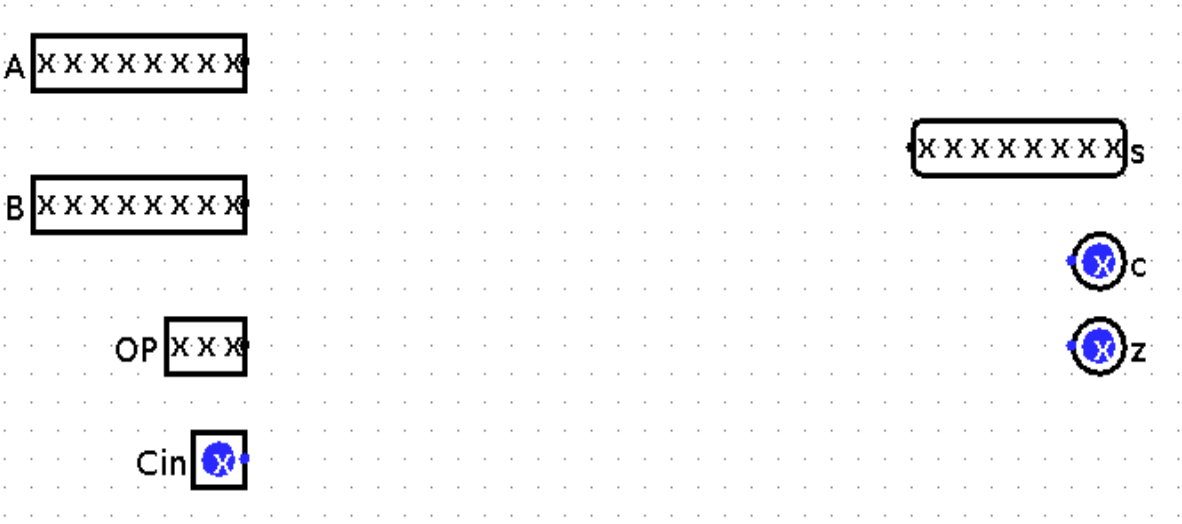

Illustration a: Entradas e saídas da ALU

Cada circuito da ALU

É muito mais simples construir um circuito para 8 bits do que 12 deles de uma só vez. É por isso que buscamos modularizar o máximo possível – e para a resolução de problemas (bugs). Somente alguém insano faria tudo de uma só vez. É claro que poderíamos poupar algumas comparações (e consequentemente performance ou dinheiro) por fazer sem a modularização, mas como o objetivo da nossa ULA é fazer apenas uma das operações de cada vez, não haverá tanto impacto.

Comparação OR bit-a-bit (A OR B):

A comparação OR é um circuito de relativamente simples implementação, principalmente por ele ser uma operação bit-a-bit. Então é um simples caso de aplicar a operação a cada bit separadamente para expandir ele para uma operação de 8 bits e então juntar em uma saída de 8 bits:

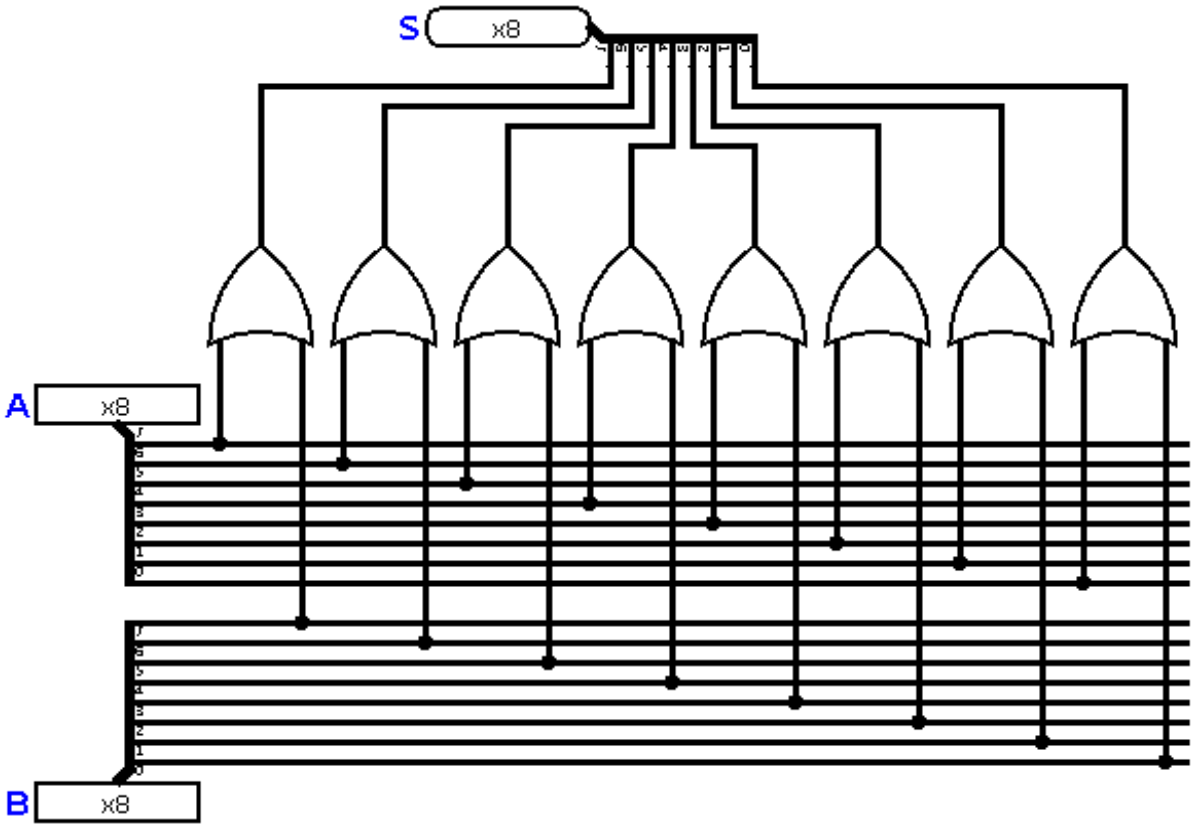

Illustration b: OR 8bits

Comparação AND bit-a-bit (A AND B)

Tal como a comparação OR, a comparação AND também é aplicada bit-a-bit, então podemos aplicar o mesmo método para essa operação e o resultado devera ser A AND B (8bits):

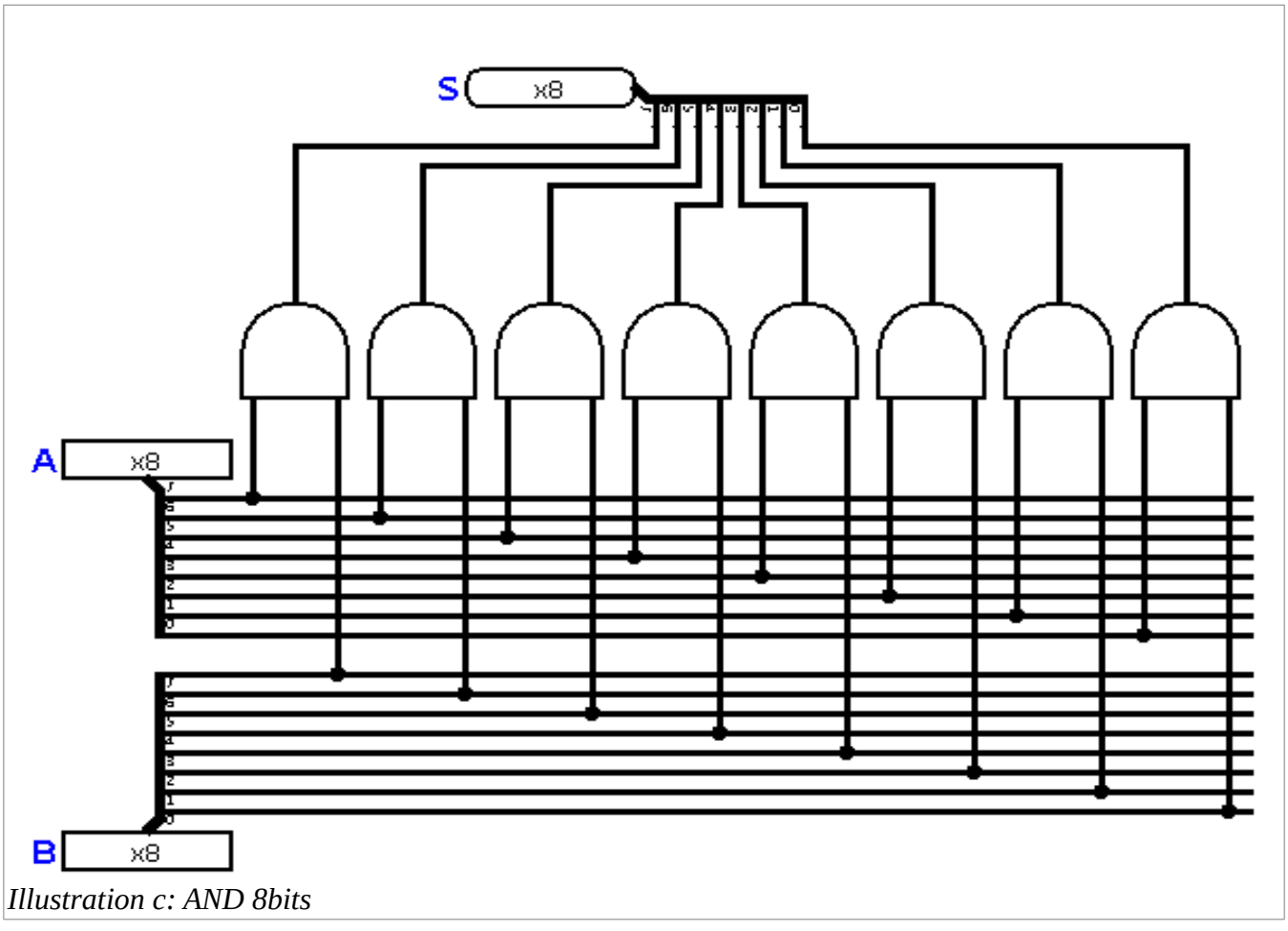

Illustration c: AND 8bits

Henrique Coutinho Layber e Renan Moreira Gomes
Universidade Federal do Espírito Santo
Vitória 2018

Comparação XOR bit-a-bit (A XOR B)

Como a comparação `XOR` (diferente de) não é uma porta básica (sendo elas a `AND`, `OR` e `NOT`) então é necessário defini-la e então modularizar e usar para fazer o `XOR` para 8 bits. Apór fazer o `XOR` de 1 bit, pode-se considerar como uma porta básica e então ela também é aplicada bit-a-bit, levando usar a mesma técnica:

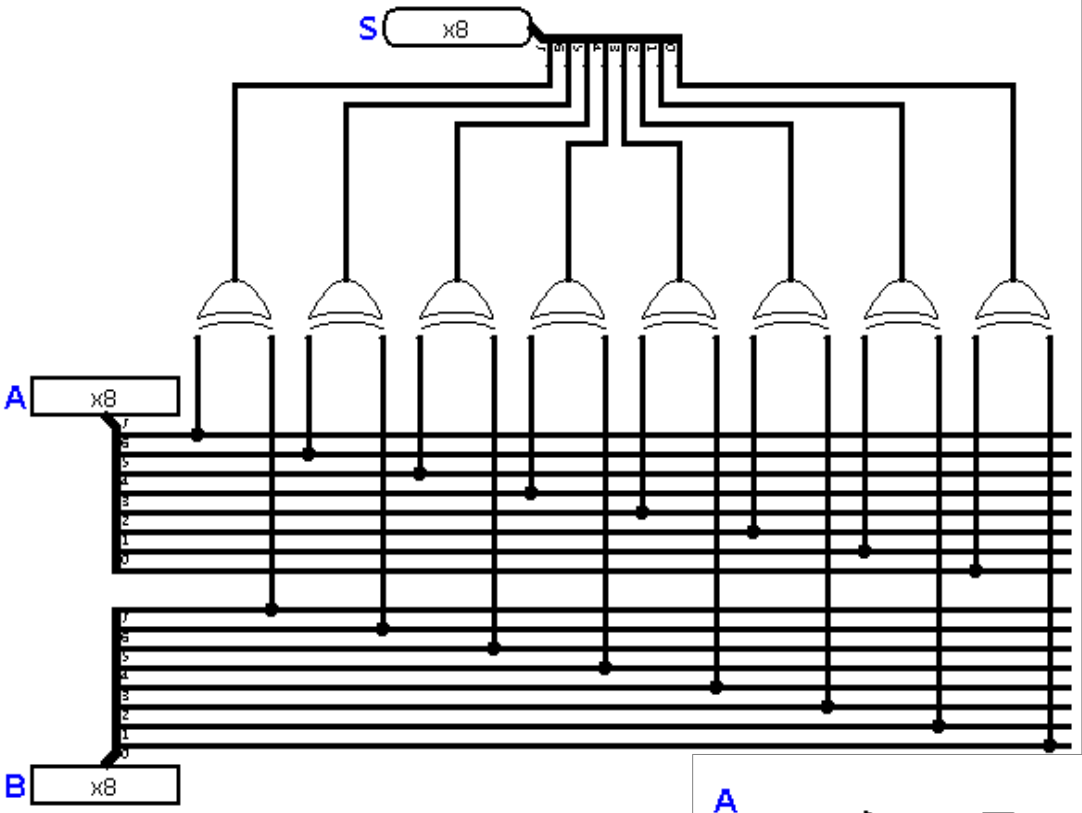

Illustration d: XOR 8bits

Cada sinal "D" é um XOR de 1 bit, representado na illustration e.

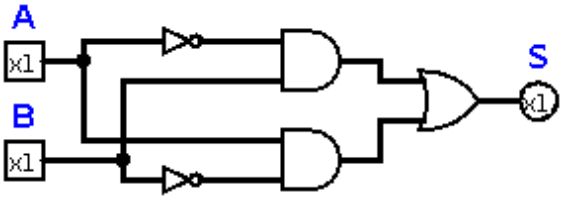

Illustration e: XOR 1 bit

Note que não é preciso ser complexo para a modularização ser útil.

ADD soma (A + B)

Para realizar a soma de 8 bits, devemos criar inicialmente um somador completo de 1 bit a partir da tabela 2, observe a imagem abaixo:

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

Table a: Tabela-Verdade do somador de 1 bit completo

Usando a tabela-verdade, fica fácil constuir um somador completo (de 1 bit), e usamos a saí Cout (`carry out`) para levar uma entrada “sobe um” de uma soma. Note que os `carrys`, dependendo da operação, podem carregar diferentes valores, com diferentes significados.

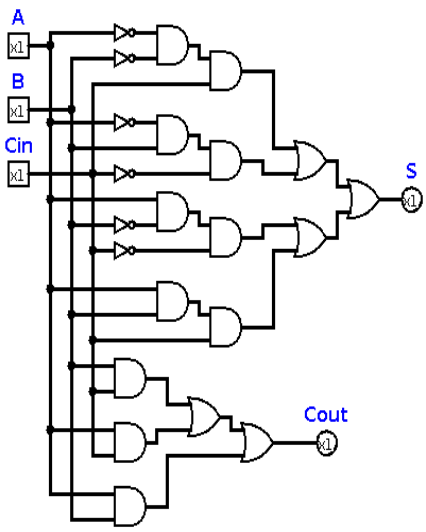

Illustration f: Somador de 1 bit completo

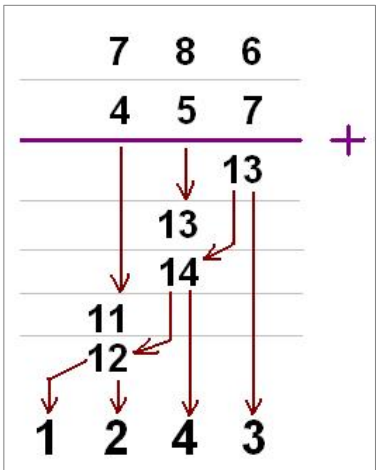

Illustration g: `Carry` de uma soma

Com o somador de um bit completo, basta encadeá-lo 8 vezes usando os `carry` para formar o somador de 8 bits:

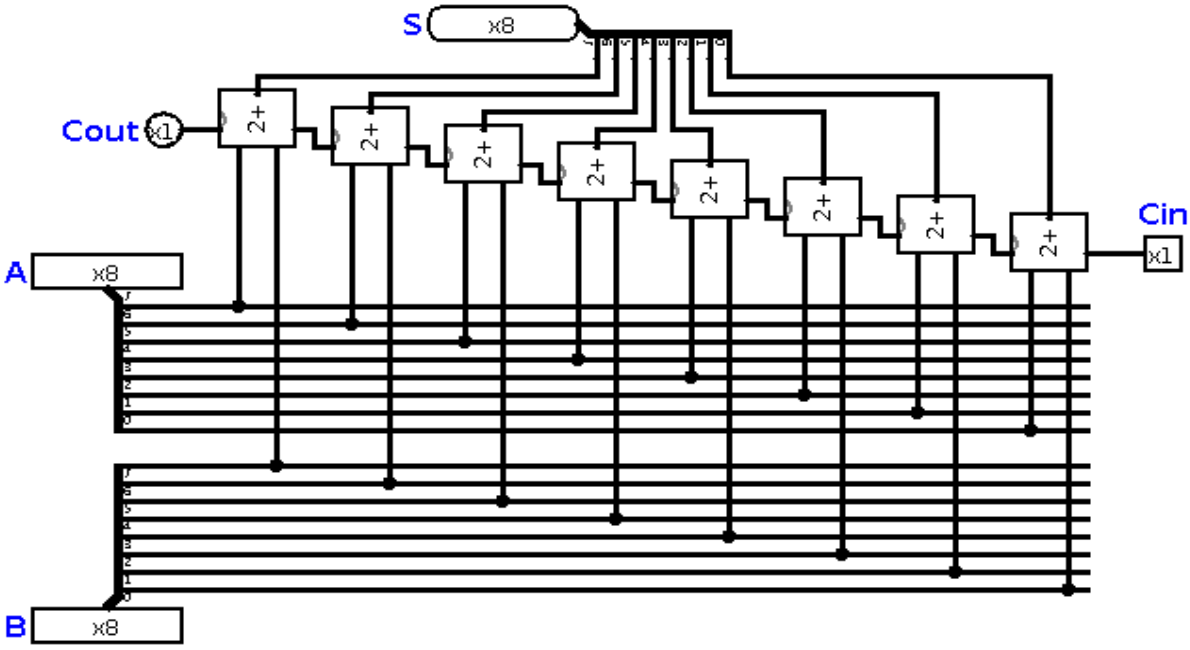

Illustration h: Somador 8 bits

E, assim, toda a *illustration g* forma o módulo de soma da ALU.

Subtração (A − B)

Irônicamente, o circuito de soma será usada para definir o circuito de subtração. Por ser feito as operações em base binária, pode-se abusar de propriedades super úteis como a representação em complemento a dois[2], e A e B poderão então serem somados se representamos B de forma negativa no complemento a dois. Para fazer isso, só é preciso inverter B (0→1 e 1→0) e somar um. Como isso seria complicado, outra estratégia é somar 1 à primeira soma, pois (2 + (-1)) +1 = (2 + (-1 + 1)).

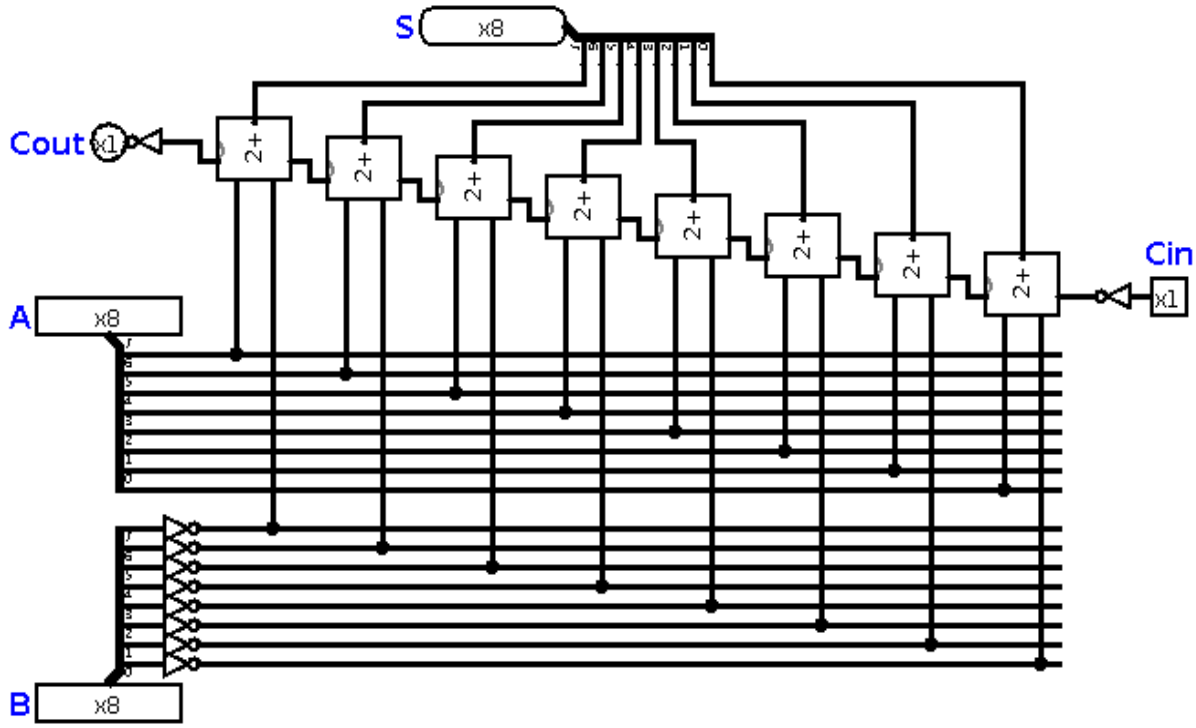

Illustration i: Subtração 8 bits

2 Complemento a dois é uma representação binária positiva e negativa, onde, diferentemente do complemento a um, não há a dupla representação do zero e nela, bem como na matemática convencional, (5) + (-2) = 3. Para isso ser válido, todos os números devem estar representados em complemento a dois (inclusive a saída).

Shift Left (SHL)

A operação Shift Left leva cada bit ao dígito à esquerda (parecido com uma multiplicação por 2), à direita entra zeros, e o bit que “cai pra fora” do tamanho especificado deve ser carregado pelo Cout, pois ao encadearmos duas ALUs, ele será passado para a próxima sessão de 8 bits (mais no cap iv. Unindo as ALUs). Então, é só uma questão de reconexão dos bits de entrada na saída:

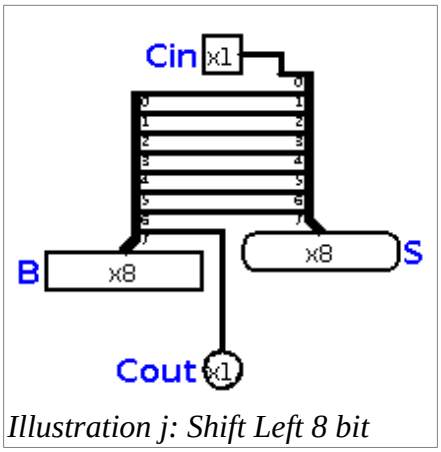

Illustration j: Shift Left 8 bit

Note que o pino 0 da saída não está ligado a nenhum bit de entrada, mas está aterrado (é o mesmo que constante `0`). Isso ocorre pois à medida que o valor se desloca a esquerda, zeros aparecem preenchendo à direita.

E também que o MSD (*Most Significant Digit* – Dígito Mais Significativo) da entrada leva à saída `carry` separada, Cout.

Shift Right (SHR)

Extendendo as possibilidades com os shifts, com o `SHR` apenas desloca todos os bits da entrada, levando o `LSD` (*Least Significant Digit* – Dígito Menos Siginificativo) para o `Cout`, e aparecendo zeros à esquerda, semelhando à um resultado de uma divisão por 2, isso pode ser facilmente e efetivamente feito for *rewiring* (religação) das entradas:

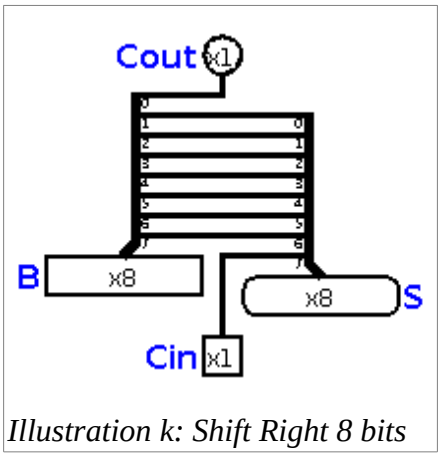

Illustration k: Shift Right 8 bits

Multiplexadores

É preciso escolher qual das operações serão realizadas, e para isso é eficiente um circuito característico denominado *multiplexador*. Eles são característicos pois agem como se fossem chaves seletoras (muitas vezes chamados assim), permitindo que um código de operação controle qual das entradas ele repete na única saída (também há o inverso – em qual das saídas ele repete a única entrada – os demultiplexadores).

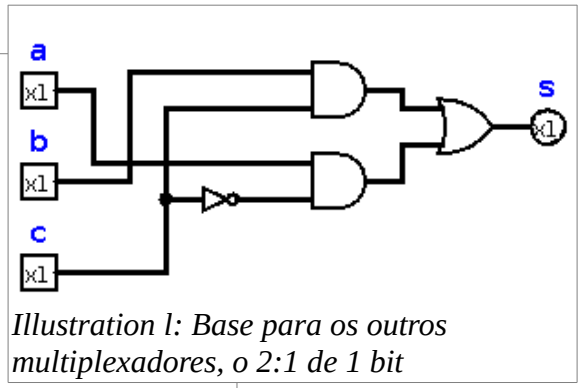

Illustration l: Base para os outros multiplexadores, o 2:1 de 1 bit

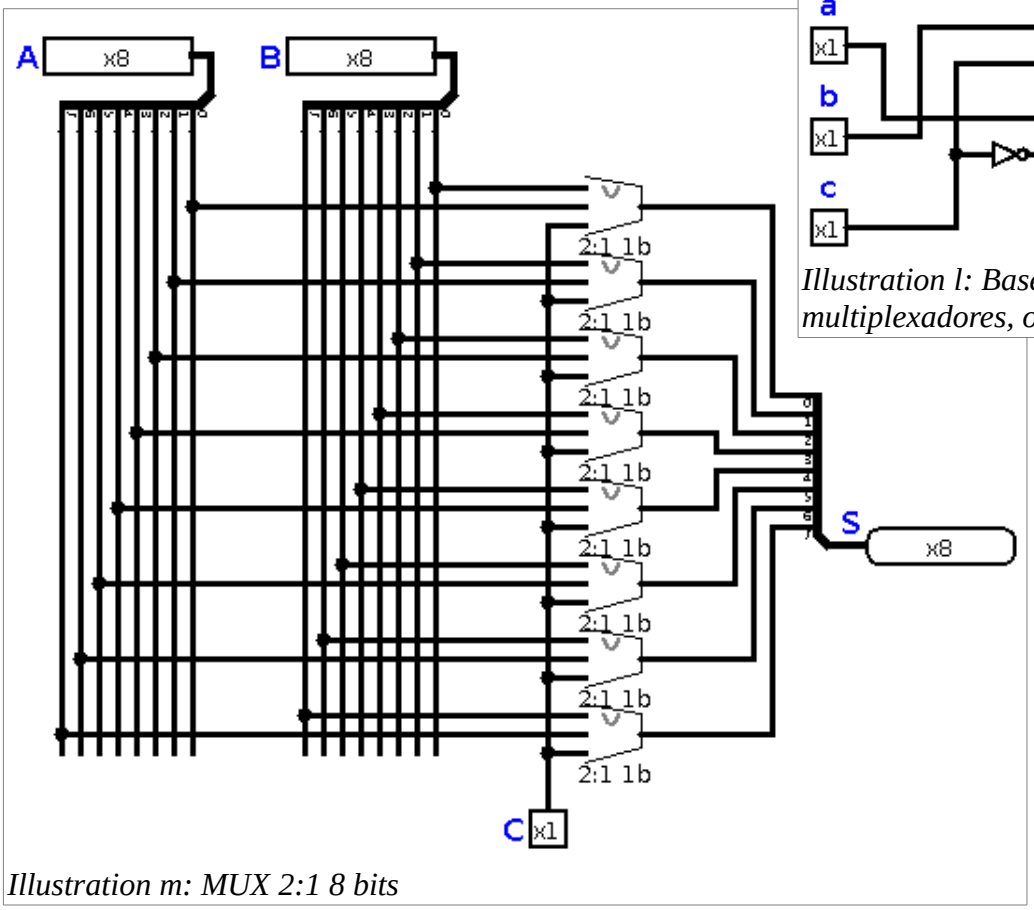

Illustration m: MUX 2:1 8 bits

Agora pode-se usar o MUX 2:1 de 8 bits para construir o MUX 8:1 de 8 bits, o que de fato selecionará a operação a ser realizada:

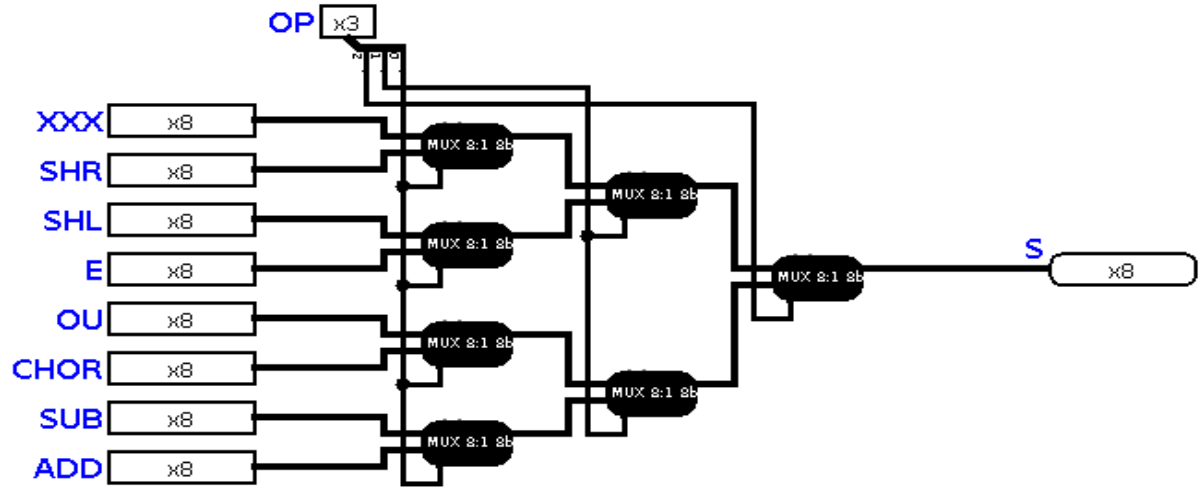

Illustration n: MUX 8:1 8 bits

Foram implementados multiplexadores conjuntos para a operação e para o `carry`, partindo do mesmo operador, de forma a somente informar que ocorreu `carry` quando essa informação é relevante. Pois como as saídas para os operadores de AND, OR, XOR e o não usado deve ser `0`, aterra-se a entrada onde o operador selecionaria esses dados:

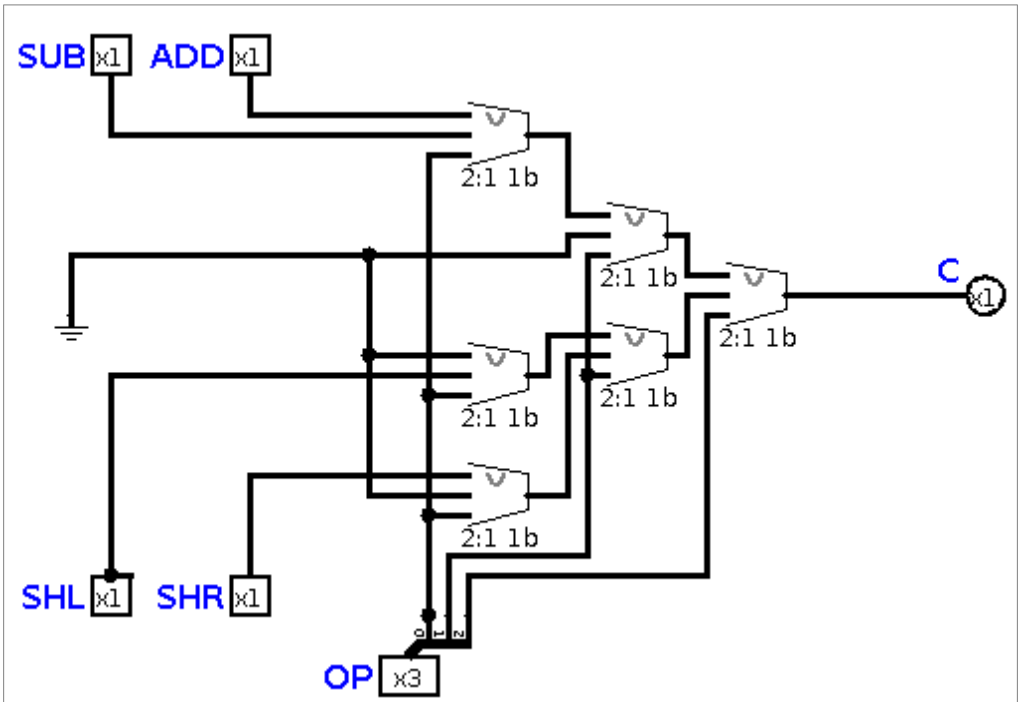

Illustration o: MUX carry (4:1 1 bit)

Igual a 0

Como o nome perfeitamente escolhido sugere, esse circuito verifica se a entrada (no caso, de `8` bits) é igual a `0`. É um tanto simples uma vez que já foi feito o circuito `AND` bit-a-bit, e conhecendo a tabela-verdade de valores do `AND`, pode-se simplesmente inverter cada bit da entrada e passar por um `AND` com ele mesmo (assim todos devem ser iguais a `1` para sair resultado `1`, porém eles foram invertidos e na realidade seriam todos iguais a `0`:

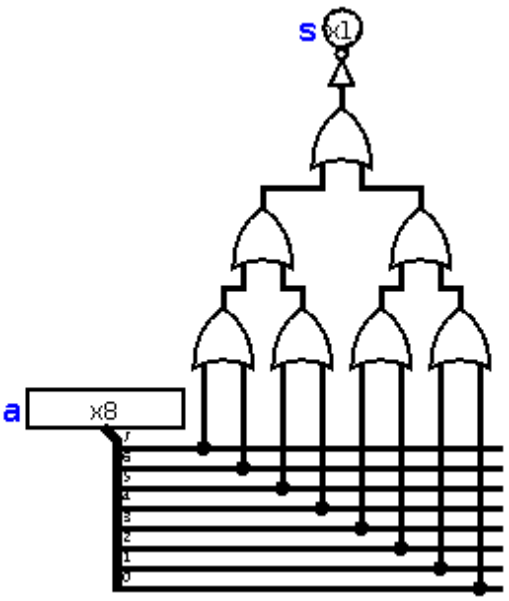

Illustration p: Igual a 0 (8 bits)

Display Hexadecimal

Ao invés de construir um circuito pre-definido pro display, ao invés disso pode ser usado multiplexadores e usá-la como linhas da tabela-verdade (dessa forma o circuito continua grande, mas legível). Então define-se um multiplexador 16:1 de 1 bit, e ele foi usado como linhas da tabela-verdade para ligar o display 7-segmentos[3]:

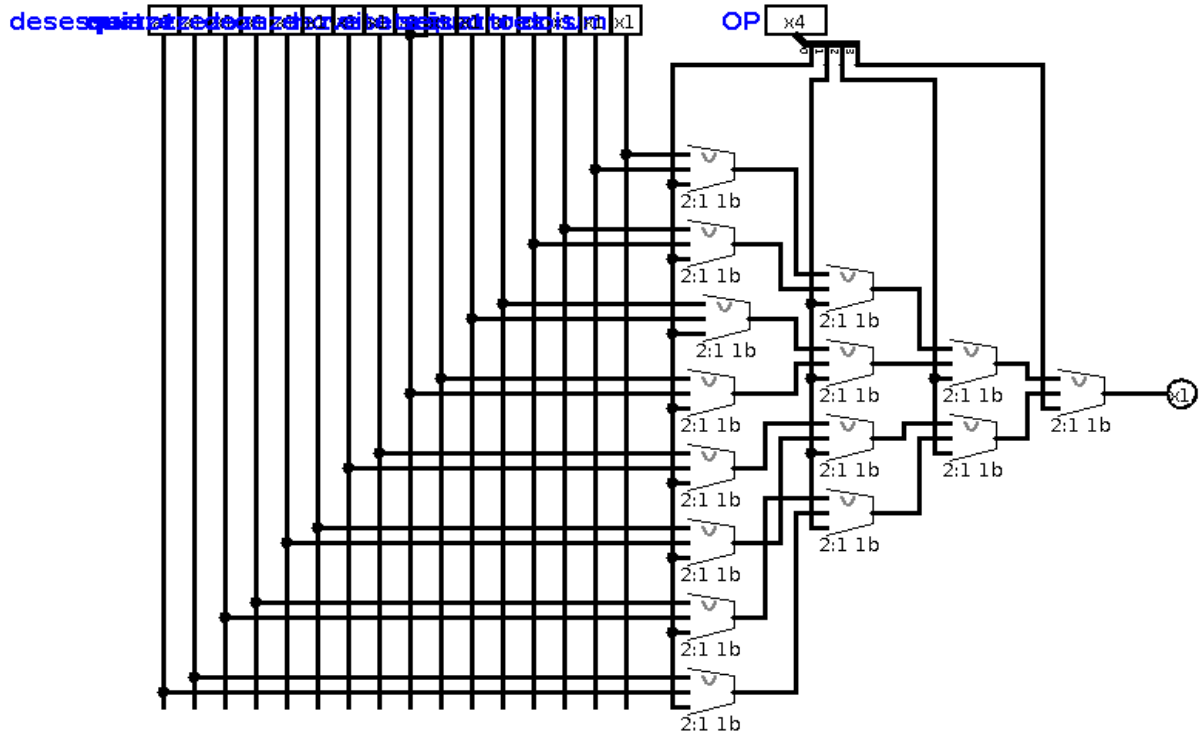

Illustration q: MUL 16:1 (1 bit)

3 O decodificador pode ser encontrado no Anexo 1. Ele foi removido do capítulo original graças ao seu tamanho e em prol da extensão e legibilidade do relatório.
São 16 entradas para cada uma das 7 saídas… 112 entradas!

Montando a ALU

Agora que temos todas as operações prontas, basta montar usando as peças já feitas: das entradas A e B, que são distribuídas de acordo com cada operação, e coletadas em um multiplexador usando a entrada de operação (Op) como chave seletora. Essa chave seletora também seleciona a saída C (`carry`), e da saída, usamos o circuito `Igual_a_0` para verificar se a saída esta igual a zero e esta é a saída `Z`. Basta montar o display, usando 4 bits de cada vez em um decodificador hexadecimal (um para cada dígito):

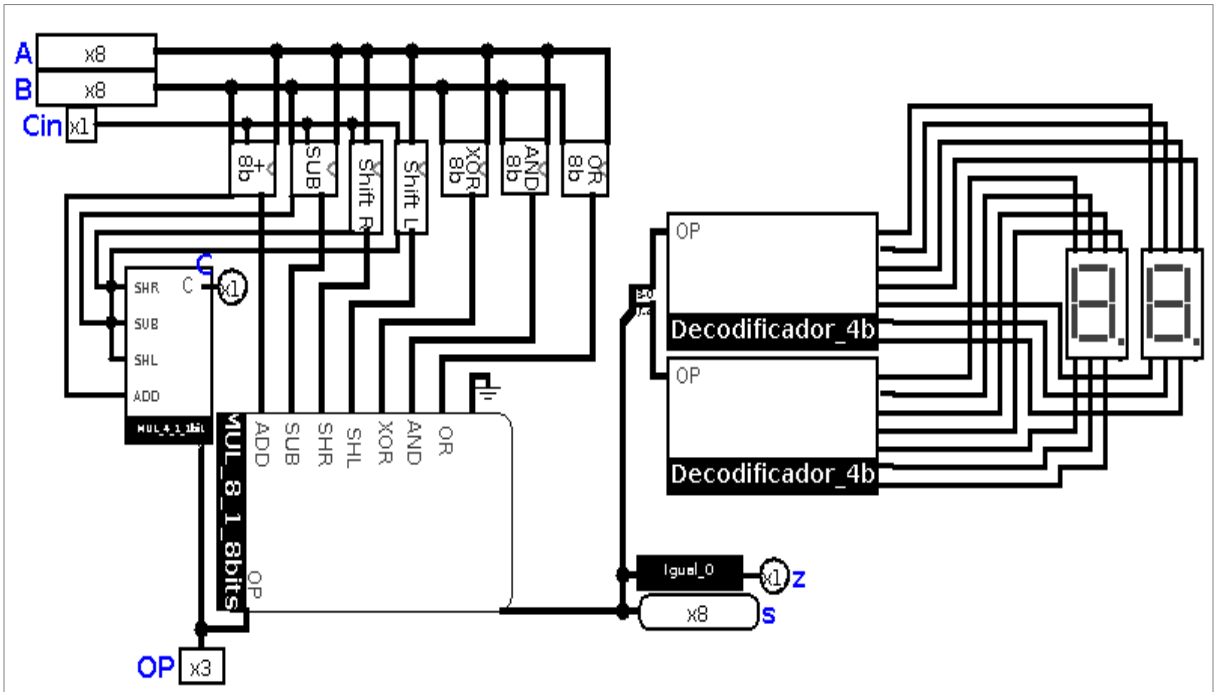

Illustration r: ALU 8 bits

Unindo as ALUs

Agora, como tudo foi montado corretamente, simplesmente ligar metade de cada entrada de 16 bits em uma ALU, concatenar os resultados e ligar os `carrys`, deve-se chegar ao resultado esperado:

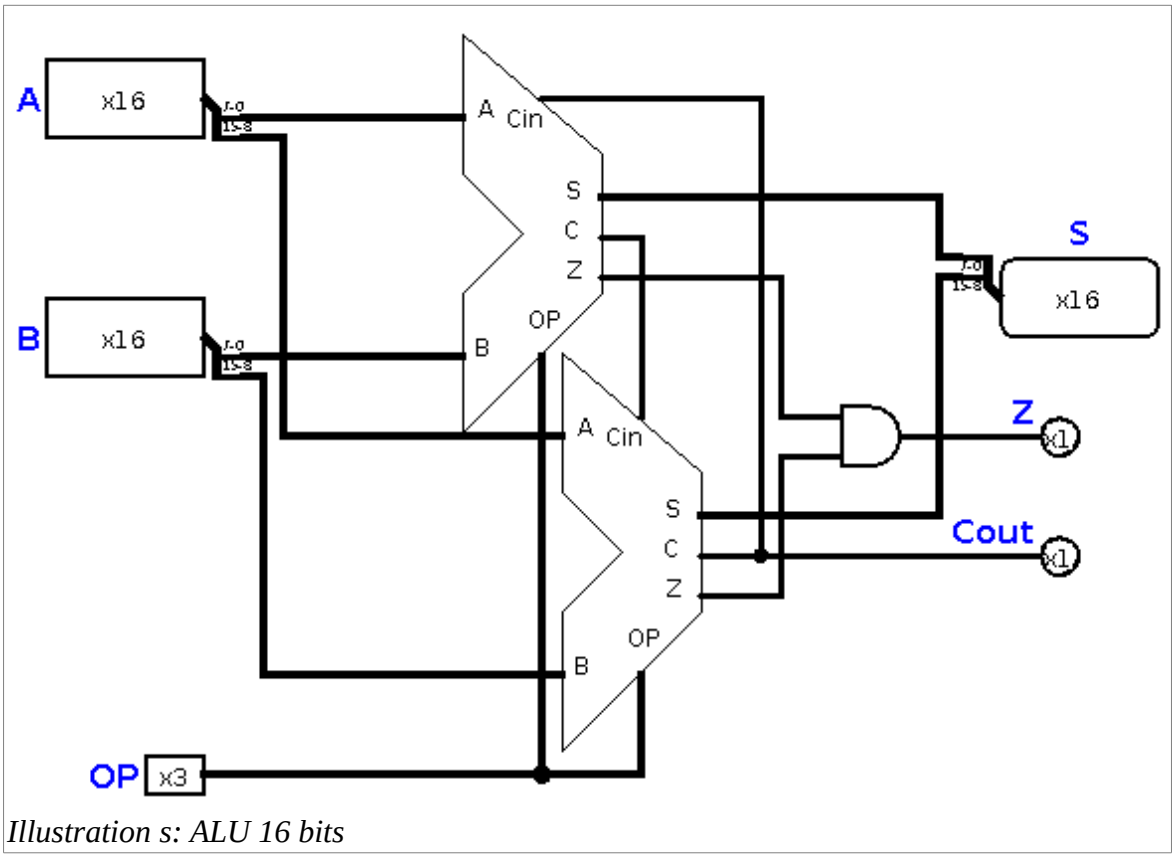

Illustration s: ALU 16 bits

ANEXO 1

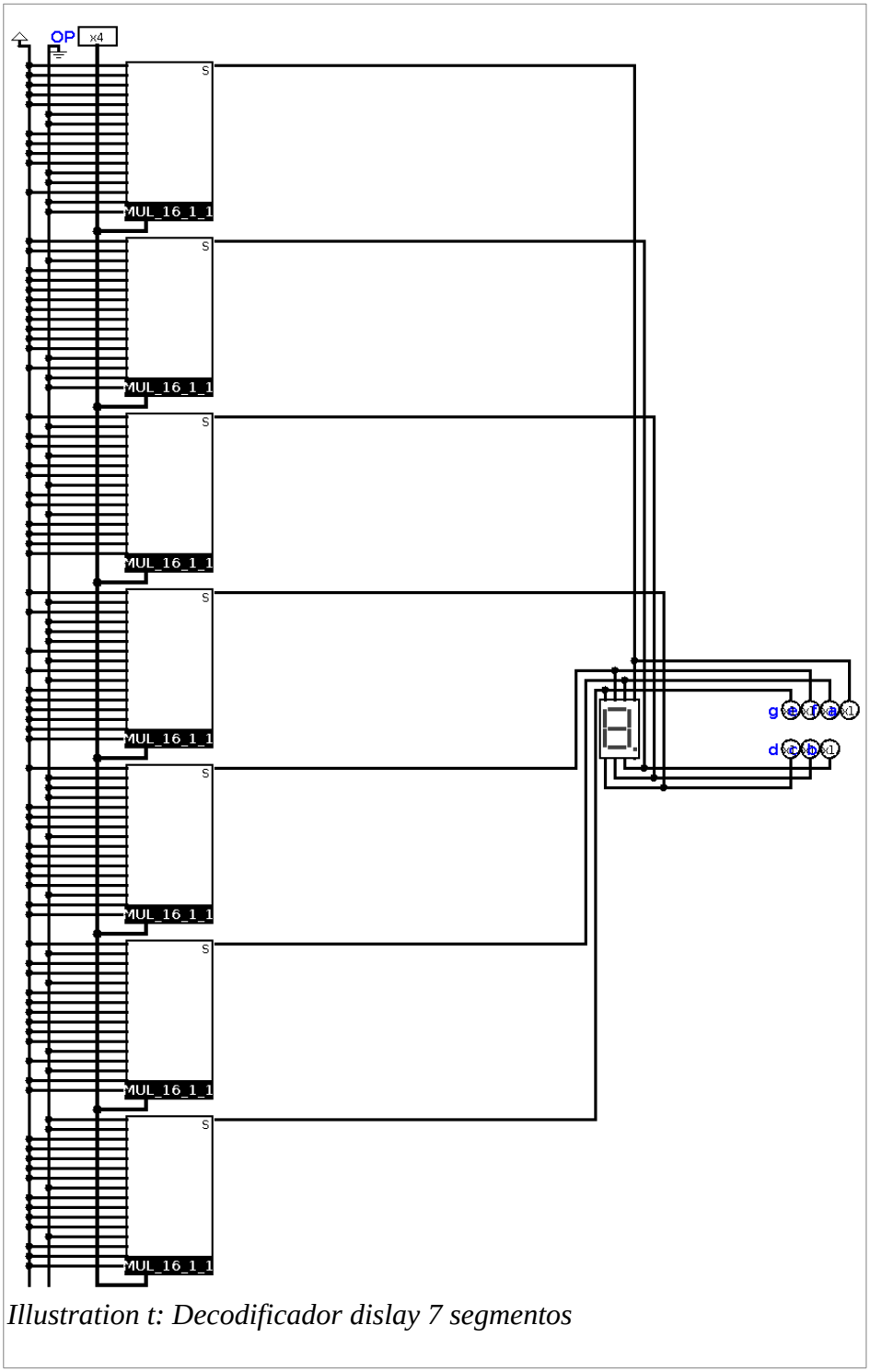

Illustration t: Decodificador dislay 7 segmentos