ISCTE IUL
Instituto Universitário de Lisboa
Departamento de Ciências e Tecnologias da Informação

Arquitectura de Computadores (I)

Textos de apoio

– Circuitos combinatórios típicos –

Draft – Versão 0.02 – Out./2009 – Fernando Batista e Tomás Brandão.

Índice

Circuitos Combinatórios Típicos

1. Introdução

Diz-se que um circuito é combinatório se, para cada combinação dos valores das entradas, os valores resultantes nas saídas são sempre os mesmos, independentemente do instante do tempo. Por outras palavras, os valores na saída dependem apenas dos valores na entrada – o circuito não tem memória.

Para ilustrar este conceito, considere um circuito com 2 entradas – A e B – e duas saídas – F e G. Ao experimentar o circuito com A=0 e B=0, o resultado obtido foi F=1 e G=0. Sendo o circuito combinatório, então pode-se garantir que, sempre que as entradas forem A=0 e B=0, o resultado nas saídas será sempre F=1 e G=0.

Considere agora um outro circuito com uma entrada A e uma saída F, que tem o seguinte comportamento: se a entrada A estiver a '0' durante um segundo ou mais, a saída é '1', caso contrário a saída é '0'. Neste caso o circuito não é combinatório, pois para o valor de entrada '0' há duas possibilidades de valores na saída – o valor de saída não depende apenas do que está na entrada.

Um circuito lógico combinatório pode sempre ser descrito através de uma tabela de verdade que relaciona directamente as suas entradas com as suas saídas. Sendo assim, todos os circuitos que foram anteriormente estudados na disciplina pertencem a esta categoria de circuitos.

Existem circuitos combinatórios que desempenham funções específicas, muitas vezes úteis no projecto de sistemas digitais de maior dimensão. Entre estes circuitos combinatórios típicos encontram-se:

- Descodificadores e codificadores;
- Multiplexers e desmultiplexers;
- Comparadores;
- Adicionadores e subtractores.

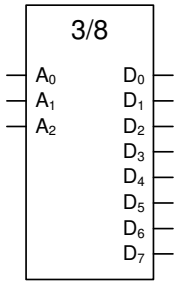
Estes circuitos típicos vão ser abordados nas próximas secções, sob o ponto de vista do seu funcionamento, construção e utilização.

2. Descodificadores

2.1. Funcionamento

Um **descodificador** é um circuito combinatório com n entradas e 2^n saídas que funciona do seguinte modo: ao ser introduzida nas entradas uma combinação binária correspondente a um número i, o descodificador irá activar a linha de saída i, ficando as restantes desactivadas.

Um descodificador com n entradas e 2^n saídas diz-se um *descodificador n para 2^n*. Por exemplo, descodificador "2 para 4" (Dec 2/4), descodificador "3 para 8" (Dec 3/8), etc. Na seguinte figura encontra-se representado um descodificador 3/8. O circuito possui 3 entradas $A_2A_1A_0$ e 8 saídas D_0 a D_7 (pois 2^3=8).



De entre as entradas A_2, A_1 e A_0, considera-se que a entrada A_2 é a correspondente ao bit de maior peso.

Suponha que a combinação binária apresentada nas entradas é a combinação "001" (ou seja, A_2=0, A_1=0 e A_0=1). Neste caso, a única saída que fica com o valor lógico '1' será a saída D_1. As restantes saídas ficam todas com o valor lógico '0'. Caso a combinação apresentada à entrada fosse "101", seria a saída D_5 a ser activada, pois $(101)_2 = 5$.

Existem outros circuitos combinatórios que habitualmente se designam por descodificadores, como é o caso do descodificador *BCD-para-7 segmentos* e do descodificador *BCD-para-Excesso-3*. Estes circuitos não são na realidade descodificadores tal como definidos nesta secção. Seria mais correcto designá-los por *conversores de código*, uma vez que transformam uma dada palavra de código numa outra representada por um código diferente.

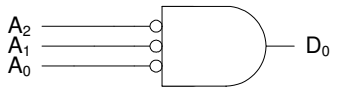
2.2. Esquema interno

Tendo em conta o que foi dito anteriormente, pode-se descrever o comportamento de um descodificador 3/8 através da seguinte tabela de verdade:

A_2	A_1	A_0	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	**1**	0	0	0	0	0	0	0
0	0	1	0	**1**	0	0	0	0	0	0
0	1	0	0	0	**1**	0	0	0	0	0
0	1	1	0	0	0	**1**	0	0	0	0
1	0	0	0	0	0	0	**1**	0	0	0
1	0	1	0	0	0	0	0	**1**	0	0
1	1	0	0	0	0	0	0	0	**1**	0
1	1	1	0	0	0	0	0	0	0	**1**

Repare agora numa particularidade: cada uma das saídas só dá resultado '1' para uma única combinação de entrada. E como já foi visto, a combinação binária de entrada que activa a saída D_i será aquela que corresponde ao valor *i*.

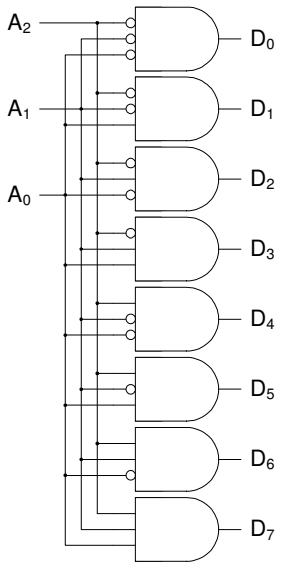
Começando pela saída D_0, por observação da tabela de verdade, verifica-se que esta só dá '1' quando a entrada é "000" (A_2=0, A_1=0, A_0=0). Ou seja, a saída D_0 é descrita pelo termo mínimo $\overline{A_2} \cdot \overline{A_1} \cdot \overline{A_0}$, o que corresponde ao seguinte circuito:



Para produzir a saída D_1, o raciocínio é semelhante: esta só dá resultado '1' para a combinação binária "001". Usando o termo mínimo correspondente, D_1 pode ser escrito como:

$$D_1 = \overline{A_2} \cdot \overline{A_1} \cdot A_0 .$$

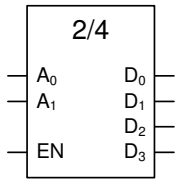
Seguindo este raciocínio para projectar as restantes saídas, chega-se à conclusão que cada uma das saídas D_i do descodificador é dada pelo termo mínimo de ordem *i* (D_i=m_i). Chega-se deste modo ao circuito representado na figura seguinte, que corresponde ao esquema interno de um descodificador 3/8.



Para descodificadores com dimensões diferentes (2/4, 4/16, etc.) o esquema interno seria em tudo semelhante, com as devidas adaptações ao número de variáveis em cada termo mínimo.

2.3. Sinais de *Enable*

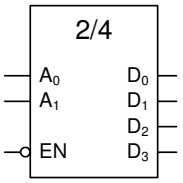
Nos circuitos combinatórios típicos, como é o caso dos descodificadores, é comum existirem entradas adicionais que permitem activar / desactivar o circuito de uma forma global. Este tipo de entradas tem habitualmente a designação de ***enable***. Nos esquemas dos circuitos, é comum utilizar as iniciais "EN" (*enable*) ou "OE" (*output enable*). Tal pode ser observado na seguinte figura, que representa um descodificador 2/4 com entrada de *enable*.



Quando o *enable* está desactivado (se estiver a '0') todas as saídas do circuito são desactivadas, ficando a '0'. Quando o *enable* está activado, i.e. EN=1, então o circuito comporta-se normalmente – faz a função de descodificador.

O sinal de *enable* pode ser visto como um interruptor do circuito: quando está activado o circuito funciona, cumprindo o propósito para o qual foi construído; quando está desactivado, o circuito não faz nada, dando sempre '0' nas suas saídas, independentemente do que se coloca nas entradas.

Note que existem circuitos nos quais o sinal de *enable* funciona do modo contrário ao descrito: o circuito activa quando EN=0 e desactiva quando EN=1. Nestes casos, a representação esquemática inclui uma "bolinha" (que representa a negação) associada à entrada de *enable*, como se exemplifica na seguinte figura:



Quando existem entradas de *enable*, o nível lógico com que este é activado encontra-se descrito no catálogo do circuito correspondente.

2.4. Construção de um descodificador a partir de outros

Existem muitas situações em que se pretende construir um descodificador de maior dimensão (isto é, com um maior número de entradas e saídas) a partir de descodificadores de menores dimensões.

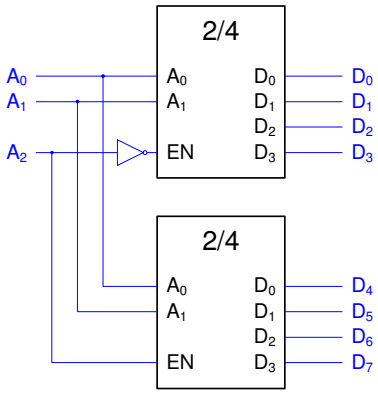
Por exemplo, no projecto de um circuito combinatório poderia acontecer uma situação em que seria útil utilizar um descodificador 3/8, mas apenas estavam disponíveis descodificadores 2/4. O problema a resolver neste caso seria então o seguinte: tendo apenas disponíveis descodificadores 2/4, como construir facilmente um circuito que se comporta como um descodificador 3/8?

Para resolver este problema é importante ter presente o modo como funciona um descodificador 3/8. Relembre que um descodificador deste tipo tem entradas $A_2A_1A_0$ (em que A_2 é a de maior peso) e saídas D_0 a D_7.

Num descodificador 3/8, quando a variável de entrada A_2 é igual a '0', então a linha de saída activada será uma das linhas D_0 a D_3 ('000' a '011'). Quando A_2 é '1', então a linha activada estará dentro do conjunto D_4 a D_7 ('100' a '111'). Sendo assim, pode-se afirmar que a variável

A_2 controla em qual dos dois grupos – linhas D_0 a D_3 ou linhas D_4 a D_7 – se encontra a linha a activar.

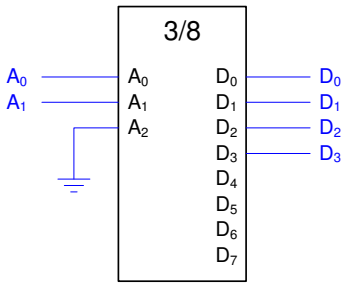
Para construir o descodificador 3/8 a partir de descodificadores 2/4, podem-se usar dois descodificadores 2/4 – um deles fica associado ao grupo de linhas D_0 a D_3 e o outro fica associado ao grupo de linhas D_4 a D_7. A activação de cada um destes descodificadores fica a cargo da variável de entrada A_2. Este comportamento pode ser facilmente implementado se os descodificadores 2/4 tiverem uma entrada de *enable*. O circuito resultante seria o representado na seguinte figura:



Quando A_2 é igual a '0', activa-se o descodificador de cima, e quando A_2='1' activa-se o de baixo. A activação / desactivação é feita através das entradas de *enable*.

Considere agora um problema que é o inverso do caso anterior: construir um descodificador mais pequeno a partir de um maior. Considere como exemplo a construção de um descodificador 2/4 a partir de um 3/8. Como já foi visto, as saídas do descodificador 3/8 podem ser divididas segundo dois grupos, cuja activação depende da variável A_2. Para se ter um circuito que se comporta como um descodificador 2/4 basta escolher um desses grupos de saídas e ligar a variável A_2 a '0' ou a '1' consoante a escolha que foi feita.

A seguinte figura ilustra este procedimento (escolheu-se o grupo de saídas D_0 a D_3 e portanto ligou-se A_2 a '0' (Gnd).



2.5. Implementação de funções lógicas

Anteriormente, foi visto que qualquer função lógica pode ser escrita na forma normalizada *soma de termos mínimos* (um OR entre os termos mínimos que produzem o resultado '1' na função). Como a cada uma das saídas de um descodificador corresponde um termo mínimo da função,

então é possível implementar qualquer função lógica utilizando apenas um descodificador e portas lógicas OR.

Começando por um caso simples, considere a função lógica de duas variáveis definida por:

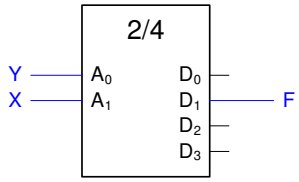
$$F = \overline{X} \cdot Y \ .$$

A tabela de verdade de F tem apenas um termo mínimo com o valor lógico '1' – o termo mínimo correspondente à combinação X=0 e Y=1, tal como se indica na tabela de verdade anexa.

X	Y	F
0	0	**0**
0	1	**1**
1	0	**0**
1	1	**0**

Para implementar a função F com base num descodificador, em primeiro lugar há que escolher a dimensão do descodificador a utilizar. Como neste caso se trata de uma função de duas variáveis, o descodificador mais adequado é um 2/4.

Para implementar o circuito, ligam-se as variáveis da função às entradas do descodificador, tendo o cuidado de preservar a ordem dos bits. Neste exemplo, como Y é a variável de menor peso, então será ligada à entrada de menor peso do descodificador. Como a função é composta apenas pelo termo mínimo $\overline{X} \cdot Y$ (que corresponde à combinação "01"), a função é obtida através da linha de saída D_1. O circuito resultante é o seguinte:



Tendo em conta o funcionamento do descodificador, o valor de D_1 (ou seja, F) só será '1' quando nas entradas A_1A_0 estiver a combinação "01" (ou seja, X=0 e Y=1). Nos restantes casos D_1 (ou F) é sempre '0'.

Considerando agora um caso ligeiramente mais complexo, suponha que se pretende implementar um circuito que realize a função

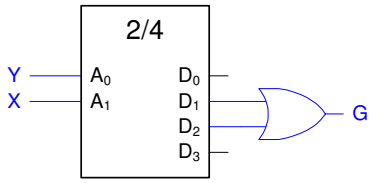
$$G = \overline{X} \cdot Y + X \cdot \overline{Y} \ ,$$

com base num descodificador 2/4.

Agora a função é composta pela soma de dois termos mínimos: o termo $\overline{X} \cdot Y$, correspondente à combinação "01" e o termo $X \cdot \overline{Y}$, correspondente à combinação "10". A função G pode ser descrita pela tabela de verdade anexa.

X	Y	F
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

Seguindo a mesma linha de raciocínio do caso anterior, a saída G é agora dada por um OR entre as linhas D_1 e D_2.



A linha D_1 está a '1' apenas no caso de estar nas entradas a combinação "01"; a linha D_2 só está a '1' quando nas entradas está a combinação "10". O OR assegura que a saída G irá ser '1' em ambos os casos.

No caso geral de uma função de *n* variáveis, para implementar a função utilizando como base um descodificador, basta juntar através de um OR as linha de saída correspondentes aos termos mínimos que produzem '1's na função.

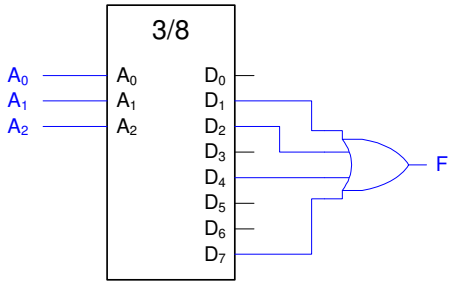
Exemplo: Com base num descodificador, pretende-se projectar um circuito que implemente a função impar para combinações de 3 bits.

Recorde que a função impar, no contexto da lógica, é uma função que dá resultado '1' para combinações com um número impar de '1's, e resultado '0' nos restantes casos. Pode por isso ser descrita pela seguinte tabela de verdade:

X	Y	Z	F
0	0	0	**0**
0	0	1	**1**
0	1	0	**1**
0	1	1	**0**
1	0	0	**1**
1	0	1	**0**
1	1	0	**0**
1	1	1	**1**

O descodificador mais adequado para resolver este problema é um descodificador 3/8, pois trata-se de uma função lógica de três variáveis. Pretende-se que a saída dê '1' para as combinações de entrada "001", "010", "100" e "111", que correspondem aos termos mínimos m_1, m_2, m_4, e m_7.

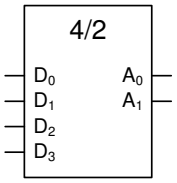
O circuito resultante será o seguinte:



2.6. Codificador

Um **codificador** é um circuito combinatório que realiza a operação inversa do descodificador. Tem 2^n linhas de entrada e tem n saídas, funcionando do seguinte modo: assumindo que apenas uma das linhas de entrada está '1', num dado instante de tempo, a saída de um codificador é a combinação binária que corresponde ao índice dessa linha. Por exemplo, num codificador 8/3, se a linha de entrada 5 estiver a '1', então o resultado nas saídas será a combinação binária correspondente ao número 5 – "101".

A seguinte tabela ilustra o funcionamento de um codificador 4/2. Note que a tabela representada não é bem uma tabela de verdade, pois não se encontra descrito o comportamento do circuito para todas as combinações de entrada. Só estão presentes os casos em que *apenas uma e só uma* das linhas de entrada D_0 a D_3 está a '1'.



D_3	D_2	D_1	D_0	A_1	A_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Os restantes casos seriam casos em que poderiam estar várias linhas activadas em simultâneo (ou mesmo nenhuma linha activada). Existem vários critérios possíveis para lidar com esses casos: dar sempre o resultado 0, ser indiferente o resultado que dá, ou dar a combinação correspondente à linha com menor índice que foi activada (ou a correspondente à linha de maior índice).

Neste último caso tem-se um circuito chamado *codificador de prioridades*, que impõe prioridades entre as diversas linhas. Por exemplo, se as linhas D_1 e D_3 se encontrassem ambas activadas, então o resultado da saída seria a combinação "01" – pode-se portanto afirmar que a linha 1 tem prioridade em relação à linha 3.

Um codificador para o qual é indiferente o valor das saídas no caso de estarem várias linhas activadas em simultâneo (ou não estar nenhuma linha activada) pode ser facilmente construído usando apenas portas OR.

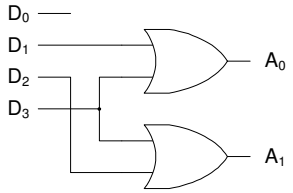
Como exemplo, será ilustrada a construção de um codificador 4/2, descrito pela tabela anterior. Com base na tabela, pode constatar-se que o bit menos significativo das saídas – A_0 – fica a '1' quando uma das linhas de entrada D_1 ou D_3 está a '1'. Sendo assim, pode-se escrever:

$$A_0 = D_1 + D_3$$

Já a saída A_1 só vai a '1' quando uma das linhas D_2 ou D_3 está activada. Fica então:

$$A_1 = D_2 + D_3$$

O circuito resultante seria o apresentado na seguinte figura:



Este circuito tem a particularidade das saídas não dependerem explicitamente de D_0.

3. Multiplexer

3.1. Funcionamento

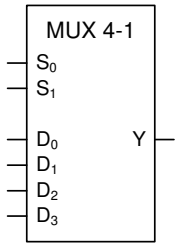
Um **multiplexer** é um circuito combinatório cuja utilidade é a de possibilitar direccionar uma das suas linhas de entrada para a saída. A escolha da linha de entrada que é direccionada para a saída é feita através de um conjunto de variáveis designadas por *selectores* ou *linhas de selecção*. As linhas de entrada do multiplexer que podem ser direccionadas para a saída designam-se geralmente por *linhas de entrada de dados*.

Um multiplexer com n linhas entradas de dados necessita de $\log_2(n)$ linhas de selecção – o que corresponde ao número de bits necessário para representar n números diferentes. Visto de outra perspectiva, um multiplexer com m linhas de selecção tem 2^m linhas de entrada de dados.

Para compreender a relação entre o número de entradas de dados e entradas de selecção basta pensar que m bits dão origem a 2^m combinações diferentes. Por exemplo, com 3 bits de selecção conseguem-se especificar 8 combinações diferentes (2^3=8). Um multiplexer com 3 linhas de selecção terá portanto 8 linhas de dados de entrada (numeradas de 0 a 7).

Um multiplexer com n linhas de entrada de dados designa-se por multiplexer "n-para-1". Por exemplo multiplexer 4-1 (MUX 4-1), 8-1 (MUX 8-1), etc.

Na figura seguinte representa-se um multiplexer 4-1. Tem 4 linhas de entrada de dados – D_0 a D_3 – e tem 2 linhas de selecção ($\log_2 4 = 2$) – S_0 e S_1 (em que S_1 será o selector correspondente ao bit de maior peso).



O comportamento do circuito pode ser descrito através da seguinte tabela (que mais uma vez não é bem uma tabela de verdade…):

S_1	S_0	Y
0	0	**D_0**
0	1	**D_1**
1	0	**D_2**
1	1	**D_3**

Quando as variáveis de selecção estão com os valores S_1=0 S_0=0, a linha D_0 é direccionada para a saída – o valor da saída será portanto aquele que estiver na linha D_0. Quando S_1=0 S_0=1, a linha D_1 é direccionada para a saída. E assim sucessivamente. A combinação binária indicada nos selectores pode ser vista como o índice da linha de entrada que se pretende direccionar para a saída.

3.2. Esquema interno

Considere o caso mais simples de um multiplexer 2-1. Este multiplexer tem 2 linhas de dados de entrada (D_0 e D_1), uma linha de selecção S e uma saída Y. Tendo em conta o funcionamento do multiplexer, quando a linha de selecção é '0', então Y = D_0, quando é '1', Y será D_1. Este comportamento pode ser descrito pela seguinte tabela:

S	**Y**
0	**D_0**
1	**D_1**

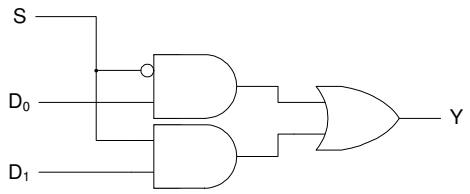
E pode também ser descrito pela seguinte expressão:

$$Y = \overline{S}D_0 + SD_1$$

Repare que, consoante S=0 ou S=1, se tem:

- para S=0: $Y = \overline{0} \cdot D_0 + 0 \cdot D_1 = 1 \cdot D_0 + 0 = D_0$
- para S=1: $Y = \overline{1} \cdot D_0 + 1 \cdot D_1 = 0 \cdot D_0 + D_1 = 0 + D_1 = D_1$

Com base na equação, o esquema interno de um multiplexer 2-1 seria o seguinte:

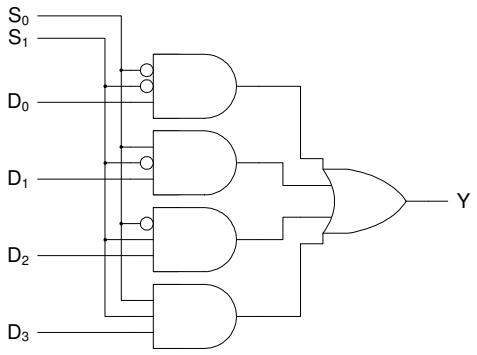


Para o caso geral, uma expressão algébrica pode ser obtida com base numa soma de produtos lógicos em que cada termo é o produto lógico entre uma entrada de dados e o termo mínimo gerado pelos selectores correspondente ao índice dessa entrada de dados.

Por exemplo, num multiplexer 4-1 a expressão da saída seria dada por:

$$Y = \overline{S}_1\overline{S}_0D_0 + \overline{S}_1S_0D_1 + S_1\overline{S}_0D_2 + S_1S_0D_3$$

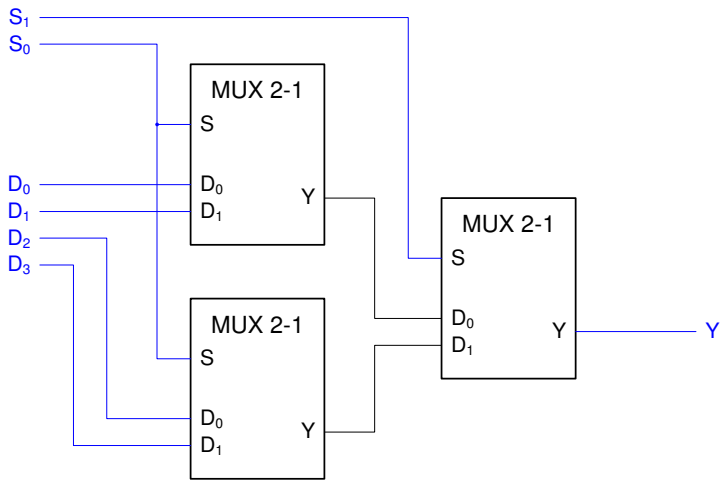
E o circuito correspondente seria:



3.3. Construção de um multiplexer a partir de outros

À semelhança do que foi visto para o caso dos descodificadores, utilizando como base multiplexers de menores dimensões podem-se construir com relativa facilidade um circuito que se comporte como um multiplexer de maior dimensão.

Tomando como exemplo a construção de um MUX 4-1 a partir de MUXs 2-1, uma possível solução é a topologia que se apresenta na figura seguinte:



Repare que existem 2 níveis de multiplexers: um nível composto por dois multiplexers, aos quais as entradas de dados estão ligadas directamente; e outro nível, composto por um único multiplexer que faz uma selecção entre as saídas do nível anterior. Os multiplexers do primeiro nível fazem uma selecção dentro de pares de entradas: entre D_0 e D_1, e entre D_2 e D_3. O multiplexer do segundo nível (e que tem maior peso) faz uma escolha entre o resultado das selecções anteriores. Repare que o selector com maior peso – S_1 – é ligado ao multiplexer do segundo nível.

3.4. Implementação de funções lógicas

Qualquer função lógica de *n* variáveis pode ser implementada recorrendo a um multiplexer com a dimensão adequada. A implementação de um circuito com base num multiplexer é simples, ligando-se directamente as entradas do circuito aos selectores do multiplexer. Quanto às linhas de entrada de dados do multiplexer, ligam-se a '0' ou a '1', consoante o resultado da função que se pretenda implementar.

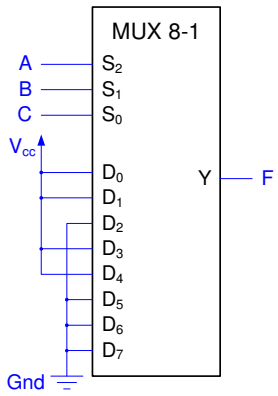
Para compreender melhor, considere que se pretende implementar a seguinte função, de três variáveis:

$$F(A,B,C) = m_0 + m_1 + m_3 + m_4$$

Como a função é de três variáveis, usa-se um multiplexer com três linhas de selecção. Como já foi discutido anteriormente, um multiplexer com 3 linhas de selecção tem 8 linhas de entrada dados (2^3=8) – usa-se portanto um multiplexer 8-1.

A tabela de verdade da função F e o circuito implementado com base no multiplexer são os seguintes:

A	B	C	F
0	0	0	**1**
0	0	1	**1**
0	1	0	**0**
0	1	1	**1**
1	0	0	**1**
1	0	1	**0**
1	1	0	**0**
1	1	1	**0**



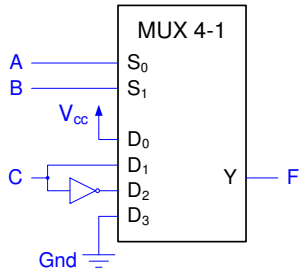
Repare na relação que existe entre o resultado da função F (na tabela de verdade) e as ligações que são feitas às entradas de dados do multiplexer D_0 a D_7. Quando se pretende a função que dê '1', liga-se a entrada correspondente a Vcc e quando se pretende que dê '0' liga-se a Gnd.

Também se consegue implementar com relativa facilidade uma função lógica de *n* variáveis, recorrendo a um multiplexer com *n-1* selectores (um multiplexer de menores dimensões). Neste caso, ligam-se *n-1* variáveis aos selectores e as entradas de dados do multiplexer passam a depender da *n*-ésima variável – a que não foi usada nas linhas de selecção.

Para ilustrar, considere que se pretende implementar a função do exemplo anterior, utilizando como base um multiplexer 4-1. A estratégia a seguir é usar duas das variáveis como selectores e fazer com que as linhas de entrada de dados dependam da terceira variável. Ligando A e B aos selectores, pode-se seguir o seguinte raciocínio:

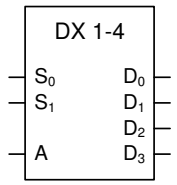
- Quando A=0 e B=0, o valor de F é sempre '1', logo liga-se D_0 a Vcc.
- Quando A=0 e B=1, o valor de F é igual ao valor de C. Sendo assim, liga-se C a D_1.
- Quando A=1 e B=0, o valor de F é igual à negação do valor de C. Sendo assim, liga-se a negação de C à entrada D_2 do multiplexer.
- Quando A=1 e B=1, o valor de F é sempre '1', logo liga-se D_3 a Gnd.

A	B	F
0	0	**1**
0	1	**C**
1	0	$\overline{\mathbf{C}}$
1	1	**0**



3.5. Desmultiplexer

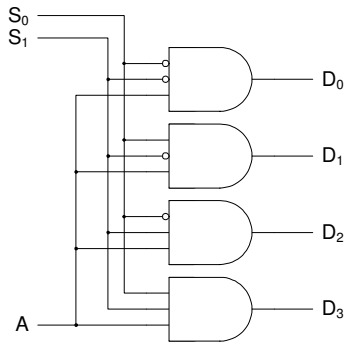
Um **desmultiplexer** é um circuito combinatório que realiza a operação inversa de um multiplexer. A operação inversa consiste neste caso em encaminhar uma entrada de dados para uma de *n* possíveis linhas de saída. As variáveis de selecção servem para escolher qual a linha de saída para a qual a entrada é direccionada.



O esquema interno de um desmultiplexer pode ser deduzido tendo em conta que, para uma dada combinação binária dos selectores, o valor lógico que vai aparecer na linha de saída correspondente a essa combinação é o valor da entrada. Por exemplo, num desmultiplexer 1-4, quando S_1S_0=00, a linha de saída para a qual a entrada é direccionada é a linha D_0. Pode-se então escrever:

$$D_0 = \overline{S_1}\,\overline{S_0}A$$

Seguindo este tipo de raciocínio para as restantes entradas, chega-se ao seguinte circuito:



Um desmultiplexer é o mesmo que um descodificador com linha de *enable*: a entrada de dados é a própria linha de *enable* e os selectores são as entradas do descodificador

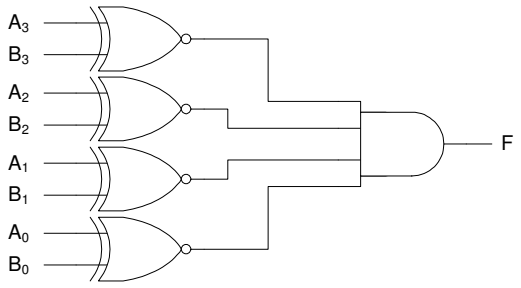
4. Comparador

Um **comparador**, na sua forma mais básica, é um circuito combinatório que dá resultado '1' quando duas combinações binárias apresentadas nas entradas são iguais, e '0' caso contrário. Admitindo que as combinações binárias a comparar são compostas por *n* bits cada uma, então um comparador terá 2*n* entradas e uma saída.

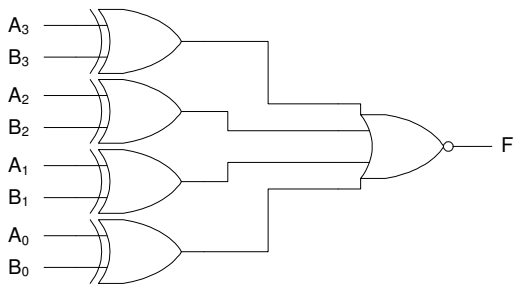
Existem também comparadores que, para além de terem uma saída que indica se as combinações são iguais, possuem também saídas que indicam se uma representa um número maior do que a outra (e/ou vice-versa). Neste caso está-se a falar de um *comparador de magnitude*.

O esquema lógico de um comparador pode ser obtido através de um raciocínio muito simples: duas combinações binárias são iguais se os pares de bits que as compõem forem iguais. Por exemplo, se considerar duas combinações de 4 bits – A e B – em que A = $A_3A_2A_1A_0$ e B=$B_3B_2B_1B_0$, essas combinações são iguais se A_3=B_3, A_2=B_2, A_1=B_1 e A_0=B_0.

Para verificar se os bits dentro de cada par são iguais ou não, a operação lógica apropriada é a operação XNOR, que dá resultado '1' quando os bits à entrada são iguais, e resultado '0' caso contrário. Para as combinações serem iguais, todos os pares de bits têm que ser iguais, logo as saídas dos vários XNORs têm que estar todas as '1'. Juntam-se portanto todas num AND, resultando no seguinte circuito:



Passando as negações (as "bolinhas") das saídas dos XNORs para as entradas do AND, chega-se ao seguinte esquema, que é uma implementação mais habitual deste tipo de circuitos:



Para implementar um comparador de magnitude, a linha de raciocínio é um pouco mais complexa. Para ilustrar, vai-se assumir que se pretende implementar um comparador de magnitude que dá saída '1' quando o número introduzido na entrada A é maior do que o introduzido na entrada B. Considere de novo que A e B são dois inteiros de 4 bits. Para verificar

se A é maior do que B, o primeiro par de bits para o qual se olharia seria o par de bits correspondente aos bits mais significativos, ou seja, A_3 e B_3. Haveria nesse caso três hipóteses:

- Se $A_3 = 1$ e $B_3 = 0$, então que A>B;
- Se $A_3 = B_3$, então ainda é preciso ver o resto da combinação;
- Se $A_3 = 0$ e $B_3 = 1$, então que B>A.

Como se está a deduzir um circuito que assinale se A>B, pode-se concluir desde já que a saída terá que dar '1' para a primeira hipótese, que corresponde ao termo $A_3\overline{B_3}$. A terceira hipótese não se vai considerar (seria para um circuito que assinalasse se B>A). Quanto à segunda hipótese, como nada se pode concluir ainda, terá que se analisar o próximo par de bits (A_2 e B_2).

Pode-se deste modo começar a escrever uma expressão para a função F, que indica se A>B:

$$F = A_3\overline{B_3} + \overline{A_3 \oplus B_3} \cdot (\text{resultado da comparação dos restantes pares de bits})$$

Para determinar o resultado da comparação dos restantes pares de bits, pode-se aplicar ao par A_2 e B_2 o mesmo raciocínio que foi seguido anteriormente, fazendo com que se chegue à seguinte expressão:

$$F = A_3\overline{B_3} + \overline{A_3 \oplus B_3}(A_2\overline{B_2} + \overline{A_2 \oplus B_2}(\text{comparação dos restantes pares de bits}))$$

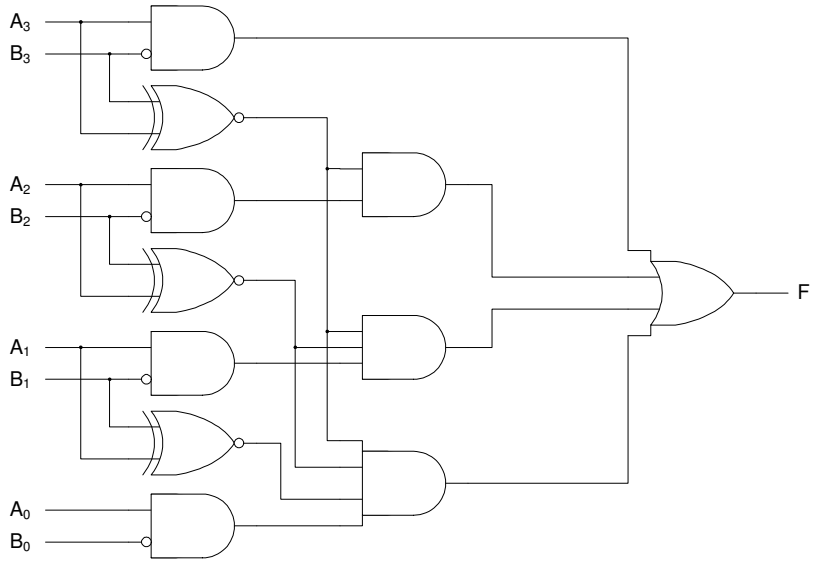
Se aplicar o raciocínio de uma forma recorrente aos restantes pares de bits em falta chega-se a:

$$F = A_3\overline{B_3} + \overline{A_3 \oplus B_3} \cdot \left[A_2\overline{B_2} + \overline{A_2 \oplus B_2} \cdot \left[A_1\overline{B_1} + \overline{A_1 \oplus B_1} \cdot \left(A_0\overline{B_0}\right)\right]\right]$$

ou, desembaraçando os parênteses:

$$F = A_3\overline{B_3} + A_2\overline{B_2} \cdot \overline{A_3 \oplus B_3} + A_1\overline{B_1} \cdot \overline{A_2 \oplus B_2} \cdot \overline{A_3 \oplus B_3} + A_0\overline{B_0} \cdot \overline{A_1 \oplus B_1} \cdot \overline{A_2 \oplus B_2} \cdot \overline{A_3 \oplus B_3}\,.$$

A essa expressão corresponde o seguinte circuito:

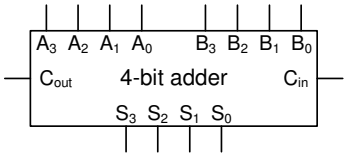


5. Adicionador

5.1. Funcionamento

Um adicionador é um circuito combinatório que calcula a soma de dois números A e B, representados com *n* bits cada um. Para além do resultado, é habitual também ser calculado o transporte dessa soma, e é também frequente haver uma entrada na qual se pode introduzir um transporte inicial.

Por exemplo, um adicionador de 4 bits é um circuito capaz de calcular a soma de dois números representados em 4 bits. Um possível esquema para um circuito deste tipo é o seguinte:



C_{in} seria uma entrada de transporte inicial da soma (que caso não se quisesse usar ligar-se-ia a '0') e C_{out} seria uma saída com o transporte resultante da soma.

5.2. Projecto de um adicionador

Recorde agora de que modo é feita a operação soma aritmética entre números representados em binário:

```
  0 1 1     ← Transportes
  1 0 0 1   ← A
+ 0 0 1 1   ← B
-----------
  1 1 0 0   ← S
```

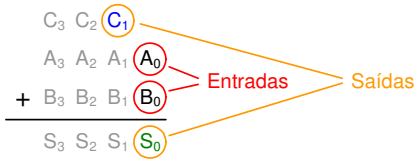
Para fazer operações deste tipo, começa-se por somar os bits menos significativos (no exemplo seria 1 + 1) dando como resultado o primeiro bit da soma e um bit de transporte que segue para a operação seguinte (0 + 1 + o '1' que vem de trás).

Uma possível abordagem para projectar um circuito que faça a soma de dois números compostos *n* bits é precisamente dividir o problema em vários "sub-circuitos", cada um deles fazendo um passo da soma.

Vai-se exemplificar a construção de um adicionador de 4 bits. Para tal, tem-se então $A=A_3A_2A_1A_0$, $B=B_3B_2B_1B_0$ e $S = S_3S_2S_1S_0$. Quanto aos bits de transporte, será designado por C_i o *i*-ésimo bit de transporte (o transporte na posição *i*). Tem-se então:

```
  C3 C2 C1    ← Transportes
  A3 A2 A1 A0 ← A
+ B3 B2 B1 B0 ← B
--------------
  S3 S2 S1 S0 ← S
```

Começando pela primeira soma que é feita – somar A_0 com B_0 – pode-se definir um circuito cujas entradas são A_0 com B_0 e cujas saídas são o resultado – S_0 – e o bit de transporte que segue para a conta seguinte – C_1:



O comportamento do circuito pode ser descrito pela seguinte tabela de verdade:

A_0	B_0	S_0	C_1
0	0	**0**	**0**
0	1	**1**	**0**
1	0	**1**	**0**
1	1	**0**	**1**

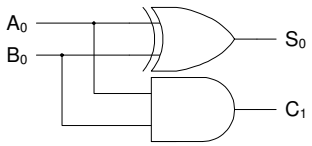
Repare que esta tabela de verdade sintetiza as várias possibilidades que podem ocorrer numa soma de dois bits:

- $0 + 0 = 0$ *e não vai nada* (C_1=0);
- $0 + 1 = 1$ *e não vai nada* (C_1=0);
- $1 + 0 = 1$ *e não vai nada* (C_1=0);
- $1 + 1 = 0$ *e vai um* (C_1=1)

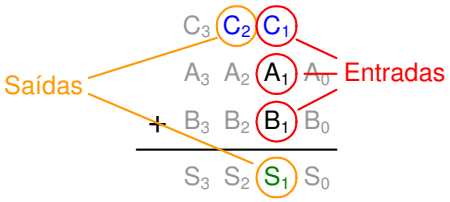
Observando a tabela de verdade de S_0 e C_1 é fácil de chegar à conclusão que as funções lógicas envolvidas podem ser definidas do seguinte modo:

$$S_0=A_0 \oplus B_0$$
$$C_1=A_0 B_0$$

O circuito correspondente tem a designação de *half-adder*:



Considerando agora o próximo passo a efectuar na conta de somar, conclui-se que o circuito a projectar para este passo tem por entradas o par de bits a somar A_1 e B_1 e um transporte que vem de trás – C_1. O circuito produz nas suas saídas a soma S_1 e o transporte C_2 que se vai passar ao próximo passo da soma.

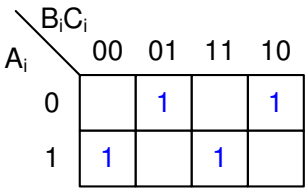


Como todos os passos seguintes vão ser idênticos a este último, pode-se generalizar o projecto do circuito, admitindo que este possui como entradas A_i, B_i e C_i, e saídas S_i e C_{i+1}. C_i é o transporte que entra na posição *i* e C_{i+1} é o transporte que segue para a posição *i*+1.

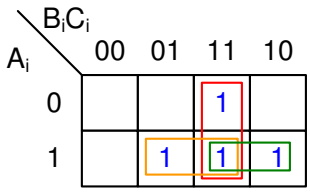
Com base no comportamento que se pretende para este novo circuito, chega-se à seguinte tabela de verdade:

A_i	B_i	C_i	S_i	C_{i+1}
0	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**

Para projectar o circuito correspondente, pode-se seguir o método habitual de projecto de circuitos – usar mapas de Karnaugh e minimizar o número de portas lógicas a usar:

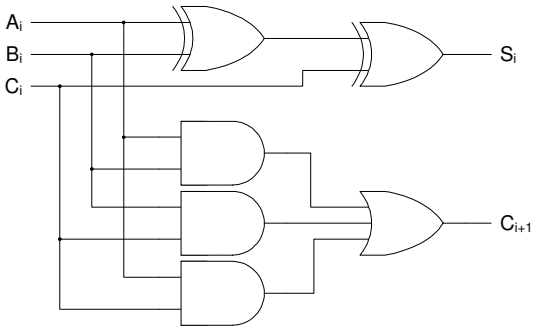


$S_i = A_i \oplus B_i \oplus C_i$

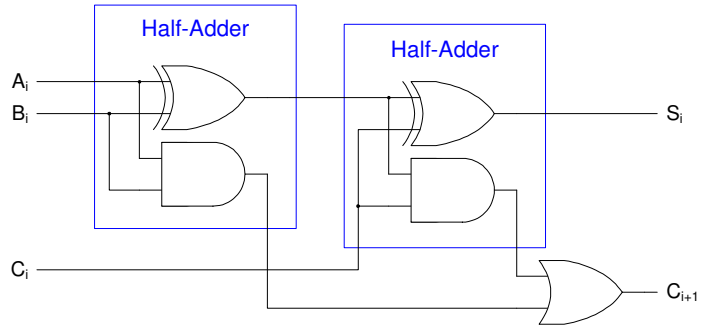


$C_{i+1} = A_iC_i + B_iC_i + A_iB_i$

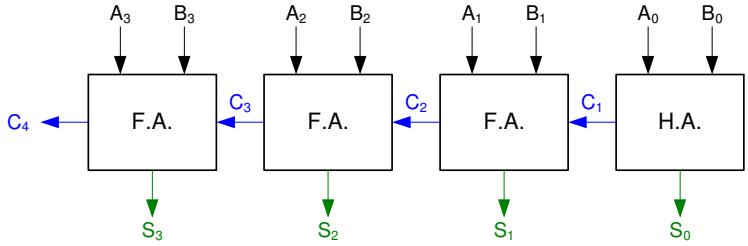
O circuito resultante, designado por *full-adder*, é o seguinte:



Outra maneira de obter um circuito *full-adder* é construí-lo com base em dois *half-adders*. A ideia é usar um *half-adder* para calcular a soma de A_i com B_i, e depois utilizar outro para somar esse resultado a C_i. O transporte que sai é '1' caso algum dos transportes destas somas feitas com *half-adders* seja '1' (será por isso um OR entre os transportes dos dois *half-adders*). Seguindo esta abordagem chega-se ao seguinte circuito:

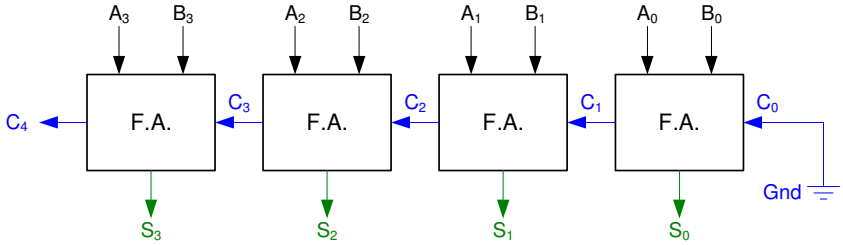


Deduzidos os circuitos *half-adder* e *full-adder*, pode-se então implementar um adicionador de *n* bits. Para tal, utiliza-se um *half-adder* para realizar a soma entre o par de bits menos significativos, fazendo-se as somas entre os restantes pares (e correspondentes transportes) utilizando *full-adders*. O Adicionador de 4 bits que resulta deste raciocínio é o seguinte:



F.A. e H.A. significam *full-adder* e *half-adder*, respectivamente. Repare também que C_4 será o bit de transporte resultante da soma completa.

Também se poderia implementar o mesmo esquema utilizando apenas *full-adders*, garantindo que o transporte que entra para o primeiro par de bits a somar é '0':



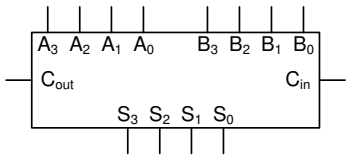
Um adicionador que segue esta topologia – full-adders ligados em série – é um adicionador do tipo *ripple-carry*. Esta topologia é simples sob o ponto de vista do funcionamento e requer uma quantidade baixa de portas lógicas para a implementação. Tem no entanto uma grande desvantagem sob o ponto de vista temporal: para se calcular o resultado da soma entre o par de bits na posição *i*, todas as somas relativas aos pares anteriores têm que ser calculadas. Como cada circuito F.A. impõe um determinado atraso temporal, o tempo que demora a ser feita uma soma entre números de *n* bits será por isso dado por $n \times T_{FA}$.

Para valores mais elevados de *n* (e.g. 64 bits), existem topologias mais vantajosas sob o ponto de vista temporal, mas que requerem mais material.

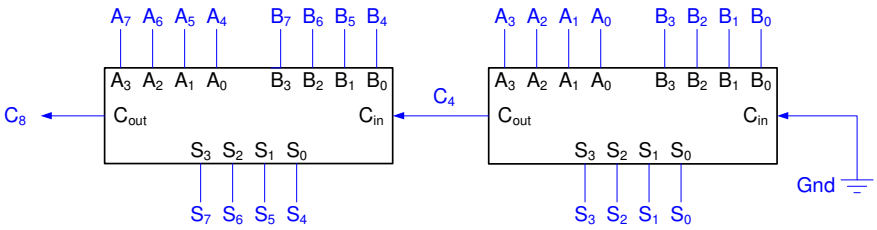
5.3. Construção de adicionadores de maior capacidade

A construção de adicionadores de maior capacidade (isto é, adicionadores que somam números representados com mais bits) utilizando adicionadores de menores dimensões é simples se for utilizada a topologia *ripple-carry*. Basta ligar vários adicionadores em série, ligando a saída de transporte C_{out} à entrada de transporte C_{in} do adicionador seguinte.

Para ilustrar, considere que se quer construir um adicionador de 8 bits a partir de adicionadores de 4 bits, idênticos ao representado na figura:



Como se pretende um adicionador de 8 bits, usam-se dois de 4 bits: um encarrega-se de somar os 4 bits menos significativos; o outro trata dos 4 bits mais significativos. O circuito resultante seria:



Para implementar adicionadores ainda maiores, o raciocínio seria semelhante.

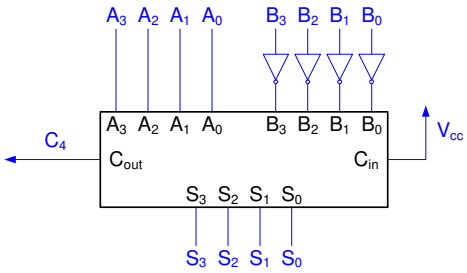
5.4. Subtracção

Subtrair pode ser encarado como "somar com o simétrico". Quer isto dizer que, desde que um sistema tenha a habilidade de calcular o simétrico, só são necessárias adições. Este é o caso dos sistemas digitais onde, de um modo geral, os inteiros e seus respectivos simétricos são representados em complemento para 2.

Sendo assim, num sistema desse género, subtrair um número é o mesmo que somar com o complemento para 2 desse número. Relembre que uma forma de calcular o complemento para 2 de um número é negar (complementar) todos bits e depois somar 1.

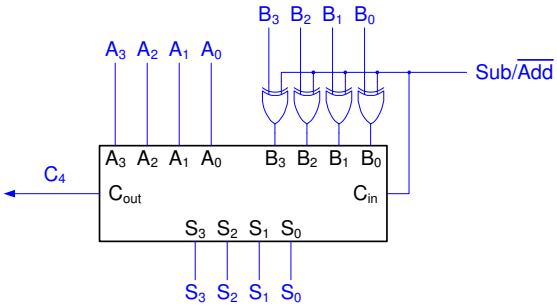
Visto desta forma, a operação S=A–B, pode ser feita de acordo com $S = A + \overline{B} + 1$ ($\overline{B}$ representa a negação de todos os bits que compõem B).

Com base nesta última expressão para S, pode-se construir um circuito que faz subtracções à custa de um adicionador e portas NOT:



As negações asseguram que todos os bits de B são complementados. A ligação de C_{in} a Vcc faz com que entre um transporte inicial a '1', que corresponde à soma de '1' no cálculo do complemento para 2.

Par finalizar, pode-se implementar um circuito que realiza tanto adições como subtracções, usando uma variável de controlo para fazer a escolha da operação a realizar:



Quando a variável de controlo ($Sub/\overline{Add}$) está a '0', entra '0' em C_{in} e os valores $B_3B_2B_1B_0$ passam para as entradas respectivas do adicionador (relembre que '0' é o elemento neutro do XOR). Deste modo soma-se A com B.

Quando a variável de controlo está a '1', entra '1' em C_{in} e os valores $B_3B_2B_1B_0$ passam negados para as entradas respectivas do adicionador (isto porque fazer XOR com '1' é o mesmo que negar). Deste modo subtrai-se B de A.

A mesma ideia podia ser estendida para somar e subtrair números representados com um maior número de bits.

6. Síntese

A seguinte tabela sintetiza de uma forma muito breve, o funcionamento dos circuitos combinatórios típicos abordados ao longo deste documento.

Circuito	Entradas	Saídas	Descrição
Descodificador	n	2^n	Introduzindo um número de n bits na entrada, activa a linha de saída correspondente. As restantes linhas ficam inactivas. **Exemplos**: dec 2/4, dec 3/8, dec 4/16, …
Codificador	2^n	n	O oposto ao descodificador. Activando uma única linha das suas entradas, obtém-se na saída o número binário correspondente a essa linha. Pode dizer-se que faz uma codificação, pois a saída será o valor numérico correspondente ao índice da linha activa. **Exemplos**: cod 4/2, cod 8/3, cod 16/4
Multiplexer	n + $\log_2(n)$ linhas de selecção	1	Dadas n linhas de entrada, o multiplexer faz com que apenas uma delas fique ligada à saída, isto é, o valor da saída será o valor da entrada seleccionada. Um multiplexer com n linhas de entrada, possui também $\log_2(n)$ linhas de selecção. Por exemplo, o multiplexer 8-1 tem 8 linhas de entrada + 3 linhas de selecção ($\log_2(8) = 3$) **Exemplos**: Mux 4-1; Mux 8-1; Mux 16-1; …
Desmultiplexer	1 + $\log_2(n)$ linhas de selecção	n	O circuito que faz a operação contrária do multiplexer. Tem uma única entrada, n saídas e $\log_2(n)$ linhas de selecção. O circuito permite direccionar a entrada para a linha indicada pelas variáveis de selecção. **Exemplos**: Dx 1-4; Dx 1-8; Dx 1-16
Comparador	$n + n$	1, 2, ou 3 (A=B, A>B, A<B)	Permite comparar 2 combinações binárias de n bits. O circuito tem uma única saída que será activa caso as combinações sejam iguais e desactiva caso contrário. **Exemplo**: Comparador 4 bits – recebe dois números de 4 bits. Tem portanto 8 entradas.
Adicionador	$n + n + c_{in}$	$n + c_{out}$	Permite fazer a soma de dois números de n bits. **Exemplo**: Adicionador de 4 bits – faz a soma de dois números de 4 bits

Lista de Revisões

Versão	Autor	Data	Comentários
001	FB	Jan/2005	Versão draft. Publicação antecipada para o período de avaliações 2004/2005
001a	TB	Jan/2005	Correcção de algumas gralhas
002	TB	Out/2009	Correcção de gralhas; Formatação do texto; Reformulação de texto; Reorganização do documento; Renovação de figuras; Introdução dos tópicos “sinais de enable”, “comparador” e “adicionador”.