4. CÓDIGOS NUMÉRICOS E OPERAÇÕES ARITMÉTICAS

Desde sempre fomos utilizando, para apoio ao cálculo automático, as inovações tecnológicas que se iam desenvolvendo para resolução de problemas do quotidiano. São exemplo mais recente as velhas máquinas calculadores baseadas em engrenagens de rodas dentadas que em função do número de dentes multiplicam, dividem, somam e subtraem. Com a proliferação dos sistemas electrónicos e por fim digitais, de imediato se passou a utilizar esta tecnologia para apoio ao cálculo aritmético. Para tal, fez-se corresponder um dígito ou algarismo da base 2 a um valor lógico. Desta forma podemos armazenar valores numéricos, e operá-los com os componentes lógicos digitais.

4.1 Representação de informação

Os valores 0 e 1 que temos vindo a referir ao longo do texto, como sendo a representação dos dois possíveis valores lógicos presentes na saída de uma porta lógica, também podem ser associados ao algarismo de um sistema de numeração binária. Um algarismo ou dígito binário é denominado por *bit* que resulta da contracção (***binary digit***).

A informação num computador é representada por um conjunto de bits que usando várias técnicas de codificação podem representar números, instruções, cores, caracteres, etc.. Por exemplo a configuração 1100101, função da codificação que se estiver a utilizar, pode representar o natural 71, o inteiro -27, o carácter **e**, etc..

4.2 Códigos numéricos

Uma vez que a lógica binária tem apenas dois símbolos (0 e 1) podemos, utilizando um sistema digital, representar números em base 2, se fizermos corresponder valores lógicos a dígitos da base 2.

4.2.1 Representação de um número

O número tal como hoje o conhecemos, é um conjunto ordenado de algarismos pesados pela sua posição relativa tendo uma correspondência biunívoca com a quantidade que representa.

Os números regem-se pela seguinte expressão de significância posicional:

$$N = \sum A_i \, B^i = A_0 B^0 + A_1 B^1 + \ldots + A_n B^n$$

Em que B é a base, A_i os algarismos dessa base e i o índice posicional do algarismo, correspondendo o índice zero ao algarismo de menor peso (algarismo das unidade). Para a base B o algarismo de maior significado é B-1. Quando a base é superior a 10, por exemplo 16, utilizam-se como algarismos, letras de A a F para representar respectivamente as configurações de 10 a 15.

Exemplo:

$(9875)_{10} = 9x10^3 + 8x10^2 + 7x10^1 + 5x10^0$

$(10110)_2 = 1x2^4 + 0x2^3 + 1x2^2 + 1x2^1 + 0x2^0$

$(7CA)_{16} = 7x16^2 + 12x16^1 + 10x16^0$

4.2.2 Conversão entre bases

Como a base de numeração que diariamente utilizamos é a base 10 (por termos 5 dedos em cada mão) para realizarmos operações aritméticas nos sistemas digitais é necessário primeiramente converter o número que se encontra na base 10 para base 2, e posteriormente converter o resultado dado pelo sistema de base 2 para base 10.

A conversão de base 10 para qualquer base obtêm-se dividindo o número decimal N sucessivamente pela base B, para a qual se quer converter até que se obtenha um quociente igual a zero como mostra o exemplo da Figura 4-1.

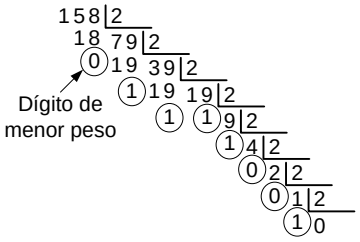


Figura 4-1

Quando se realiza a primeira divisão de N por B, o quociente dá-nos o número de vezes que a quantidade B cabe em N e o resto da divisão a quantidade sobejante, pelo que se conclui, que o resto desta divisão corresponde ao dígito de menor peso (dígito das unidades) da nova representação, produzindo as restantes divisões os dígitos de peso seguinte, pelo que:

$$158_{10}=10011110_2$$

A conversão da base B para a base 10 obtêm-se calculando a expressão de significância posicional do número N na base B, utilizando nas somas e multiplicações a efectuar a tabuada decimal.

Exemplo:

$2132_4 = 2x4^3 + 1x4^2 + 3x4^1 + 2x4^0 = 128_{10}+16_{10}+12_{10}+2_{10}= (158)_{10}$

$101101_2 = 1x2^6 + 0x2^5 + 1x2^4 + 1x2^3 + 0x2^1 + 1x2^0 = 32_{10}+8_{10}+4_{10}+1_{10}= (45)_{10}$

A conversão entre bases diferentes de 10 implicaria saber a tabuada da multiplicação e da soma na base destino. Por esta razão utiliza-se sempre a base dez como intermédia. Tomemos como exemplo a conversão de um número na base 8 para base 2

$$236_8 = 2x8^2 + 3x8^1 + 6x8^0 = 128 + 24 + 6 = 158_{10}$$

Donde se conclui que $236_8 = 158_{10} =10011110_2$

No caso em que a base para a qual se pretende realizar a conversão é potência inteira da base original, a conversão pode fazer-se por agrupamento de dígitos da base menor.

Por exemplo a conversão do número 10011110_2 para base 4, 8 ou 16, utilizando as representações descritas na Tabela 4-1, pode ser obtida da seguinte forma:

10_01_11_10 = 2132_4
10_011_110 = 236_8
1001_1110 =$9E_{16}$

Base 4	Base 8	Base 16
01=1	010=2	1001=9
10=2	011=3	1110=E
11=3	110=6	

Tabela 4-1

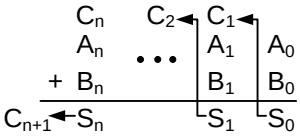
Esta é a razão pela qual, a maioria das linguagens de programação, suportam a especificação de números em base 8 (Ex: 0237 java/C) e base 16 (Ex: 0x9e java/C), pois permitem representar com menos dígitos números/configurações com uma correspondência directa à base 2.

4.3 Operações Aritméticas

Os sistemas de numeração prestam-se a algoritmos aritméticos muito simples, formalmente idênticos em qualquer base.

Operação adição

Considere a adição binária entre dois números: An,...A1,A0+Bn,...B1,B0



Ao adicionarmos dois dígitos, existe *Carry* (arrasto), se o resultado da adição desses dígitos ultrapassar o maior dígito dessa base.

Se os dígitos somados forem de peso n, o arrasto produzido tem o peso n+1, pelo que será somado aos dígitos de peso n+1, tal como fazemos em decimal, em que o arrasto da soma dos algarismos das unidades é somado aos algarismos das dezenas.

Assim sendo, cada um dos algarismos do resultado é dado pela expressão:

$S_n = |A_n + B_n + C_n|_{base}$ (o sinal + indica soma aritmética).

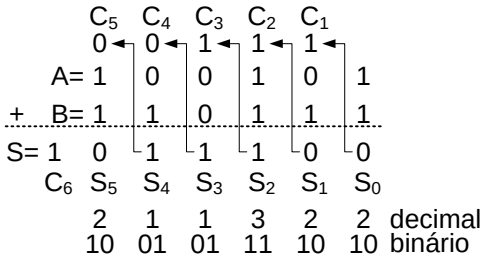
Existe C_{n+1} se $(A_n + B_n + C_n) \geq base$

Na Tabela 4-2 é apresentada a tabuada da soma da base 2.

0+0= 0
0+1= 1
1+1=10

Tabela 4-2

Considere a adição dos números 100101_2 + 110111_2



Se fizermos corresponder cada um dos algarismos da base 2 a valores lógicos podemos extrair da tabuada da soma de dois bits as seguintes expressões booleanas para $\mathbf{S_n}$ e $\mathbf{C_{n+1}}$ função de $\mathbf{A_n}$, $\mathbf{B_n}$, através da Tabela 4-3.

A_n	B_n	S_n	C_{n+1}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Tabela 4-3

$$S_n = A_n.\overline{B_n} + \overline{A_n}.B_n = A_n \oplus B_n$$

$$C_{n+1} = A_n.B_n$$

Este módulo é denominado por semi-somador e pode ser sintetizado como mostra a Figura 4-2.

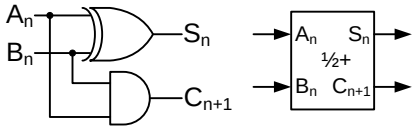


Figura 4-2

Se pretendermos implementar um circuito capaz de somar dois números de 32 bits cada, para produzir um resultado a 32 bits mais *carry*, como acontece actualmente nas unidades aritméticas dos processadores, não é trivial um solução, que passe por extrair a expressão de cada bit do resultado, função dos vários bits dos dois operandos. Iremos adoptar neste caso uma solução iterativa.

4.3.1 Solução iterativa

A solução iterativa consiste em desenvolver um módulo replicável que possa ser aplicado aos bits de um ou mais operandos. Este elemento, que denominaremos por célula, para além do bit de cada operando terá sinais de entrada e saída para diálogo com os módulos vizinhos como mostra a Figura 4-3. Esta é a solução adequada quando o número de entradas é elevado e o procedimento associado a cada entrada é recorrente. São exemplo de aplicação, as operações: soma, subtracção, comparação, etc.

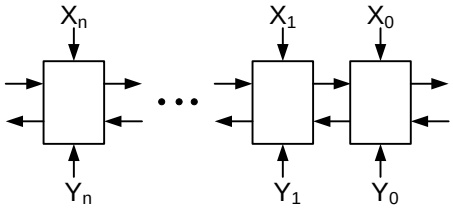


Figura 4-3

4.4 Somador completo

Como podemos observar na Figura 4-4, para responder ao algoritmo da soma de dois números constituídos por n algarismos, é necessário uma célula que possa somar três algarismos e que denominaremos por somador-completo (*full adder*). A primeira célula poderá ser um semi-somador (*half adder*), porque a soma se inicia sem arrasto prévio.

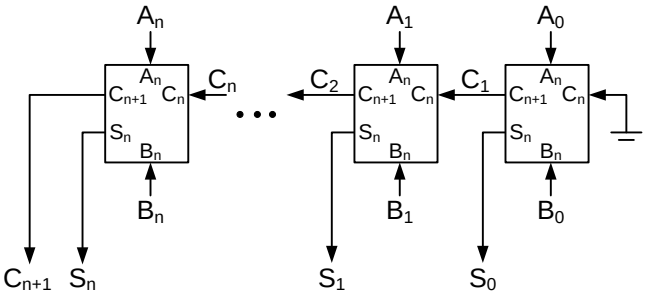


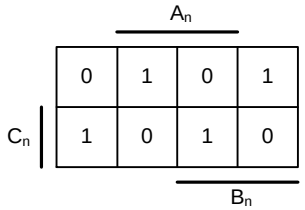
Figura 4-4

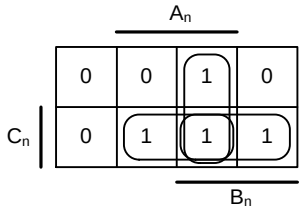
Da soma de três bits resulta a seguinte tabela de verdade:

C_n	B_n	A_n	S_n	C_{n+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

Tabela 4-4

Tabela 4-4 podemos extrair a seguinte expressão para S_n e C_{n+1}:





$$S_n = A_n \oplus B_n \oplus C_n \qquad C_{n+1} = A_n.B_n + A_n.C_n + B_n.C_n$$

Como mostra a Figura 4-5, também podemos obter uma estrutura para o somador completo por inferência modular, isto é, se tomarmos a estrutura semi-somador como componente, dado que a soma goza da propriedade associativa A+B+C=(A+B)+C (+ soma aritmética).

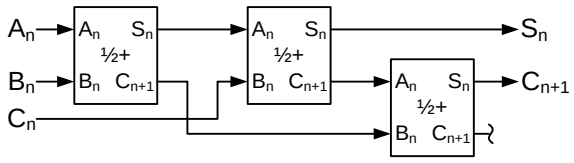


Figura 4-5

Uma vez que ao somarmos três dígitos o resultado é dado por dois dígitos, conclui-se que o terceiro semi-somador nunca produzirá arrasto, pelo que poderemos substituir o terceiro semi-somador por um XOR ou um OR, obtendo-se assim a estrutura da Figura 4-6.

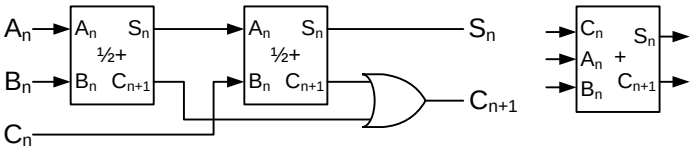


Figura 4-6

Da estrutura da Figura 4-6 conclui-se que:

$C_{n+1} = (A_n \oplus B_n).C_n + A_n.B_n$

Como mostra a Figura 4-7, utilizando os mapas de *Karnaugh* podemos concluir que esta expressão de C_{n+1} é equivalente à anteriormente obtida.

$C_{n+1} = A_n.B_n + A_n.C_n + B_n.C_n = (A_n \oplus BC_n)C_{n-1} + A_n.B_n$

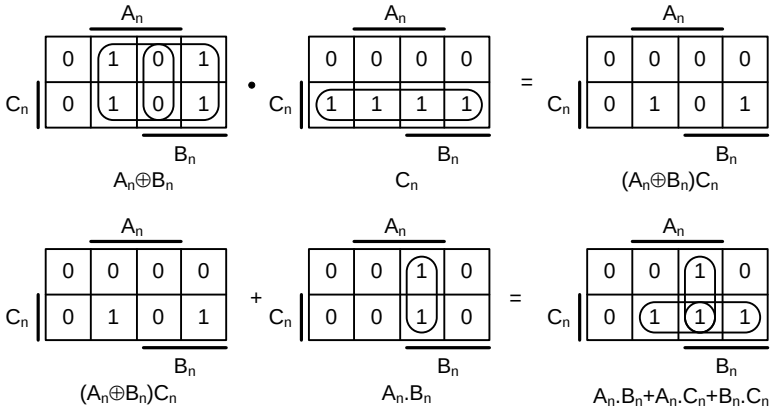


Figura 4-7

Na estrutura iterativa descrita na Figura 4-4, o tempo total de cálculo de uma soma de dois números de n algarismos é de $S_t * n$ (S_t – tempo de um somador, n – número de algarismos do número) que para certos objectivos pode constituir um sério problema. Este tempo de cálculo deve-se à propagação do *carry* através das várias células somadoras. Existem estruturas alternativas, designadas de *carry look ahead*, que calculando o arrasto separadamente, conseguem diminuir o tempo total da soma, embora utilizando mais lógicas. Para melhor compreendermos esta estrutura, vamos separar a produção da soma da propagação do *carry* como mostra a Figura 4-8. A estrutura de propagação de *carry*, utiliza a expressão alternativa de geração de *carry* e que é dada por:

$C_{n+1} = (A_n \oplus C_n)C_{n-1} + A_n.B_n.$

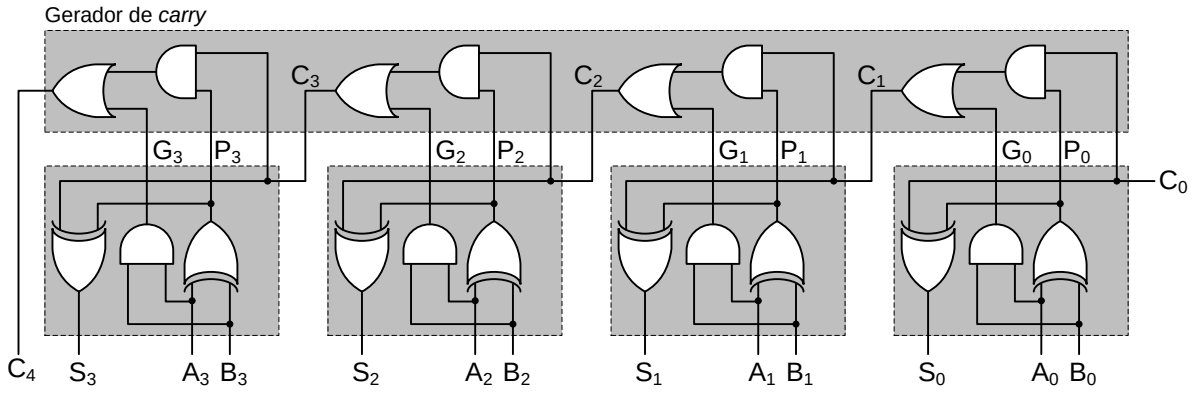


Figura 4-8

Se observarmos a estrutura que gera o *carry*, podemos extrair as seguintes expressões para cada um dos *carries*.

$C_4=G_3+G_2P_3+G_1P_2P_3+G_0P_1P_2P_3+C_0P_0P_1P_2P_3$
$C_3=G_2+G_1P_2+G_0P_1P_2+C_0P_0P_1P_2$
$C_2=G_1+G_0P_1+C_0P_0P_1$
$C_1=G_0+C_0P_0$

Obtendo-se assim a estrutura da Figura 4-9, que calcula cada um dos *carries* com apenas o atraso de propagação duas portas lógicas, o que leva a obtermos o resultado da soma após a propagação de quatro portas lógicas.

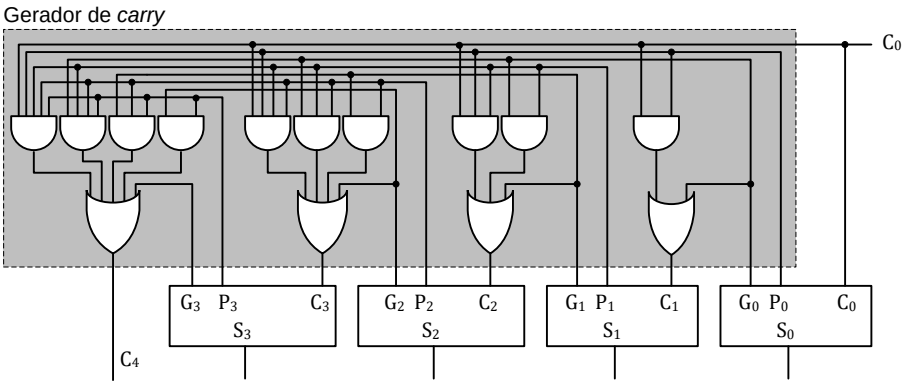


Figura 4-9

Dada a complexidade em número de portas para realizar cada uma das células, existem disponíveis no mercado circuitos integrados contendo um somador de dois números de vários algarismos, com arquitectura *carry look ahead* e com possibilidade de concatenação. São exemplo o 74HCT283 somador de dois números de 4 dígitos com entrada de *carry* de peso zero e saída de *carry* de peso 4 como mostra a Figura 4-10.

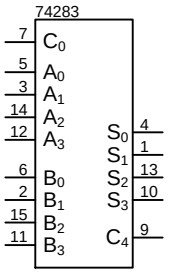


Figura 4-10

4.5 Implementação de funções utilizando somadores

Os módulos somadores são estruturas adequadas à implementação de funções ditas simétricas. Considere-se a título de exemplo, a implementação de uma função de sete variáveis, que se pretende que tome o valor lógico 1 quando o número de entradas activas for múltiplo de três. A função diz-se simétrica por ser irrelevante quais das variáveis de entrada tomam o valor 1, interessando somente quantas tomam esse valor. Dado o elevado número de entradas, a implementação desta função recorrendo à estratégia até aqui estudada (observação exaustiva de todas as combinações numa tabela de verdades) não é a mais adequada. Se considerarmos que cada entrada é um número constituído por um algarismo, ao somarmos todos as entradas poderemos observar o resultado da soma em função deste determinar se o valor é múltiplo de três. Neste caso, a função é verdade quando o resultado é 3 ou 6.

A Figura 4-11 mostra a implementação da função utilizando somadores completos de 3 bits.

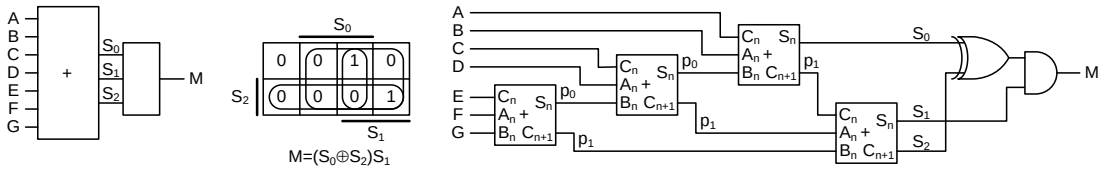


Figura 4-11

4.6 Operação Subtracção

Tal como na soma, há que entender primeiramente o algoritmo da subtracção antes de podermos encontrar um sistema digital capaz de efectuar subtracções entre dois números de vários bits.

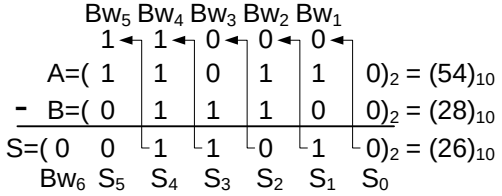


Figura 4-12

Como se pode ver na Figura 4-12, quando se subtraem dois algarismos em que o algarismo do diminuendo/aditivo é inferior ao algarismo do diminuidor/subtractivo, para que a subtracção se realize é necessário adicionar ao algarismo do diminuendo o valor da base, dizendo-se que existiu *borrow* (pedido de empréstimo), isto porque esta quantidade foi pedida aos algarismos de maior peso do número. Ao subtrairmos os algarismos de peso seguinte podemos: subtrair o *borrow* ao diminuendo, ao resultado, ou adicioná-lo ao diminuidor.

O algoritmo mais comum, por nós utilizado, quando realizamos subtracções, é o de somarmos o arrasto ao diminuidor por ser o algoritmo mais simples.

$$\begin{array}{r} 45 \\ -27 \\ \hline 18 \end{array} = \begin{array}{r} 40+5 \\ -20+7 \\ \hline \end{array} \text{ Impossível } 5<7 = \begin{array}{r} 30+15 \\ -20+\ 7 \\ \hline 10+\ 8 \end{array} = \begin{array}{r} 40+15 \\ -30+\ 7 \\ \hline 10+\ 8 \end{array}$$

O mais comum

Vejamos então a tabuada da subtracção para a base 2.

A_n	B_n	S_n	BW_{n+1}
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Tal como aconteceu para a soma, se quisermos sintetizar uma célula, capaz de ser concatenada, para com ela podermos realizar a subtracção entre dois números de vários algarismos, então esta terá que ter a estrutura apresentada na Figura 4-13.

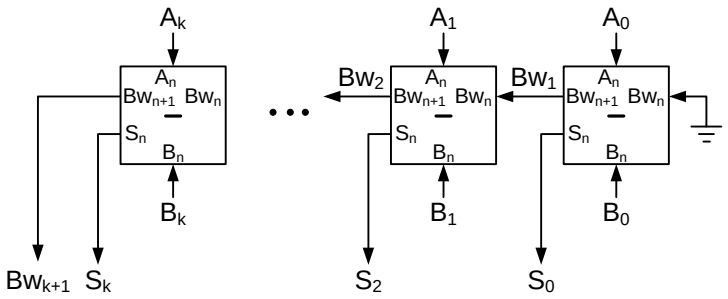
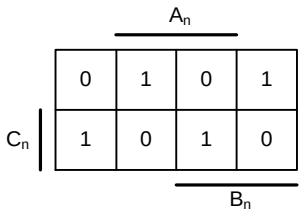
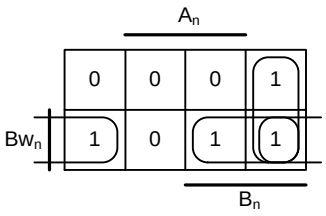


Figura 4-13

Para realizar $S_n = A_n - B_n - Bw_n$ produzindo Bw_{n+1} teremos a seguinte tabela

Bw_n	A_n	B_n	S_n	Bw_{n+1}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	1





$$S_n = A_n \oplus B_n \oplus Bw_n \qquad Bw_{n+1} = \overline{A}_n . B_n + \overline{A}_n . Bw_n + B_n . Bw_n$$

4.6.1 Subtracção inteira

Quando se subtraem dois números, pode acontecer que o resultado não seja possível representar no conjunto dos números naturais. Isto acontece, quando o minuendo é menor que o subtraendo.

Por esta razão constitui-se um outro conjunto denominado $\mathbb{Z}$, ou dos inteiros relativos, onde para cada elemento de $\mathbb{Z}$ existe o seu simétrico. O simétrico do inteiro ***a*** é algo que somado com ***a*** tem como resultado o valor zero. A forma vulgar de representar o simétrico de ***a*** é **–*a***, denomina-se esta representação por sinal e magnitude.

Desta forma, podemos utilizar o algoritmo da subtracção anteriormente estudado e caso seja produzido *borrow* pelo bit de maior peso, trocam-se os operandos, e volta-se a realizar a subtracção atribuindo o sinal (–) ao resultado. A representação deste novo conjunto $\mathbb{Z}$ num sistema digital, implica acrescentar um bit que passa a funcionar como sinal do número podendo ser considerado para o sinal (+) o bit 0 e para o sinal (–) o bit 1. Este bit tal como acontece com o sinal (+) e (-) é acrescentado do lado esquerdo do número.

4.7 Código dos complementos

A representação dos números inteiros (conjunto $\mathbb{Z}$) utilizada pelos computadores actuais não é a de sinal e magnitude. A representação utilizada é a denominada por código dos complementos. Este código é baseado no pressuposto de que os operandos e os resultados operados por um computador têm sempre o mesmo número de bits e daí que, encontrar o simétrico de um número é encontrar uma configuração que somada com ele seja igual a zero para o número de bits estabelecido, ou seja, o código dos complementos é modular, em que o módulo é igual a dois levantado ao número de bits (2^n).

Por exemplo, se estabelecermos 3 algarismos decimais para representar o conjunto dos inteiros, o simétrico de 235 é o número 765, ou seja, 765 é o complemento (acrescento necessário) a 235 para atingir o valor 1**000** (10^3), o que nos leva a concluir que para o conjunto das possíveis combinações constituídas por três algarismos existem 500 que são positivos e 500 que são negativos. Nesta representação, considera-se a primeira metade da sequência, o conjunto dos

números positivos e a segunda metade, os números negativos, o que leva a considerar o zero como número positivo.
Quando esta representação é aplicada aos números da base 2, dizemos que o número está representado em código dos complementos para 2 (2'complement). Na base 2 constata-se que se obtém o simétrico de um número (complemento verdadeiro) se somarmos o valor 1 ao complemento restringido (inversão bit a bit) do número *N*.
O complemento restringido de $A=011010010_2$ é o número 100101101_2, que corresponde à inversão de todos os bits de *A*. Se adicionarmos o valor 1 a uma das representações obtêm-se o simétrico do outro e se somarmos *A* com o simétrico de A o resultado será zero.

$$\begin{array}{rl} 100101101 & \overline{A} \\ \quad\quad +1 & \\ \hline 100101110 & -A \end{array} \qquad \begin{array}{rl} 011010010 & +A \\ +100101110 & -A \\ \hline 000000000 & \end{array}$$

Considerando o dígito mais à esquerda (maior peso) como bit de sinal, o dígito 1 representa o sinal (–), o dígito zero o sinal (+).

Exemplo:
Como representaremos então o número $-(23)_{10}$ em código dos complementos para 2?
O número 23_{10} é igual a 10111_2 tratando-se de um número natural não tem simétrico, mas se acrescentarmos zero à esquerda 010111 passamos a representar o +23. Para obter o -23 basta calcular o simétrico de 010111 que é 101000+1=101001.
Outra forma de obter a representação de -23 em código dos complementos para dois é utilizando a definição de código dos complementos anteriormente apresentada, e que é a seguinte: para representar 23 são necessários no mínimo cinco bits, acrescentando um bit para o sinal então o número mínimo de bits para representar o -23 serão seis bits. Como $2^6=64$, logo 41 é o complemento que falta a 23 para atingir 64. A representação de 41 em binário é 101001_2 que é igual à obtida da forma anterior.

Complementos
Podemos então dizer que existem dois tipos de complementos para qualquer base *b*: o complemento verdadeiro e o complemento restringido. O primeiro é normalmente denominado de complemento para a base e o segundo como complemento para a base -1 (*b-1*).
Para qualquer número *N* em base 2 com *n* dígitos, o complemento restringido é definido como $(2^n-1) - N$.
Vejamos então em binário, 2^n é um número iniciado pelo dígito 1 seguido de n zeros, logo 2^n-1 é um número binário constituído por *n* dígitos com valor 1. Por exemplo para $n = 4$, $2^4 = (10000)_2$ e $2^4-1 = (1111)_2$. Assim sendo, em binário, o complemento restringido (2^n-1) do número *N* é obtido subtraindo cada dígito de *N* ao dígito 1. Quando se subtrai um dígito binário ao dígito 1 são geradas apenas duas situações: 1 – 0 = 1 e 1 – 1 = 0, o que leva o dígito original mudar de 1 para 0 e de 0 para 1.

$$\begin{array}{rl} & 1\ 1\ 1\ 1\ \ (2^4-1) \\ - & N_3N_2N_1N_0\ \ N \\ \hline & \overline{N}_3\overline{N}_2\overline{N}_1\overline{N}_0\ \ \overline{N} \end{array}$$

Podemos assim concluir, que o complemento restringido de *N*, corresponde a inverter cada um dos dígitos de *N*.
O complemento verdadeiro de um número *N* em binário é dado por 2^n-N e que corresponde ao simétrico de *N*. Tomando a expressão do complemento restringido [(2n-1) – *N*] poderemos

expressar o complemento verdadeiro da seguinte forma 2^n-$N = [(2^n - 1) - N] + 1$, que corresponde a inverter os dígitos de N e somar o valor 1.

4.7.1 Subtracção utilizando complemento para 2

Com o novo conjunto $\mathbb{Z}$ podemos realizar a subtracção de dois números A e B da seguinte forma:

$$A - B = A + (-B)$$

Como o simétrico/complemento verdadeiro de B é dado $\overline{B} + 1$, a subtracção de A e B pode ser dada pela expressão $R = A + \overline{B} + 1$ (+ sinal de soma aritmética), o que nos permite realizar a operação de subtracção utilizando o módulo somador como mostra a Figura 4-14.

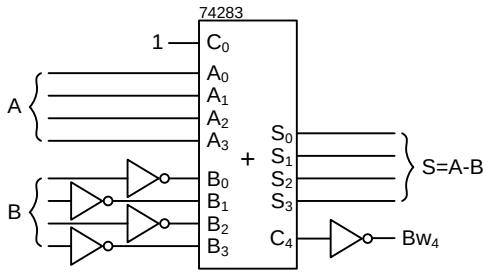


Figura 4-14

Quando se realiza a subtracção de dois números A e B tomados como números naturais utilizando a forma $A + \overline{B} + 1$, podemos verificar que o arrasto final, neste caso C_4, produzido pela soma é igual ao inverso do *borrow* da subtracção Figura 4-15, razão pela qual é necessário inverter a saída C_4, no somador como mostra a Figura 4-14.

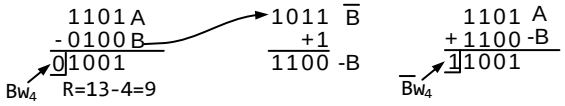


Figura 4-15

Como a subtracção de números binários utilizando o operador soma, baseia-se na aplicação da identidade:

$$A - B = A + (2^n - B) - 2^n$$

sendo (2^n – B) o complemento verdadeiro de B, então poderemos escrever:

$$A - B = A + [(2^n - 1) - B] + 1 - 2^n = (A + \overline{B} + 1) - 2^n$$

que corresponde a subtrair 2^n ao resultado da soma de A com o simétrico de B, ou seja, subtrair o valor 1 a C_4 e dai a inversão de C_4 para produzir o Bw_4.

A razão desta inversão pode ser vista ainda na seguinte perspectiva: Ao dizermos que estamos a realizar a subtracção de A por B, pela soma de A com $-B$ utilizando o código dos complementos, podemos tomar A e B pelo seu valor absoluto acrescentando um zero à esquerda como mostra a Figura 4-16, assim sendo, ambos operandos passam a representar números positivos com *n+1* bit. Caso o bit de peso n do resultado, seja 1 (resultado negativo) indica que A é menor que B, ou seja, visto em números naturais existiria *borrow*.

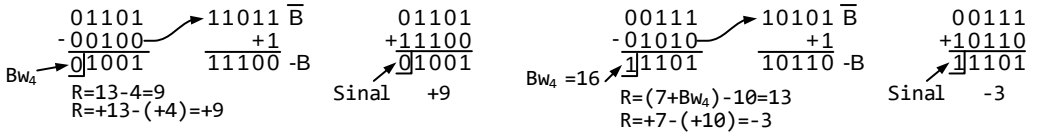
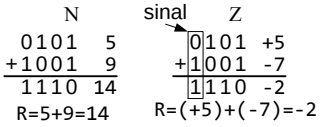


Figura 4-16

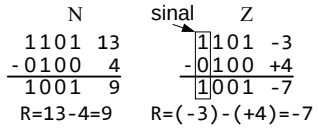
Tanto para a adição como para a subtracção, os operandos *A* e *B* podem codificar um valor natural ou um inteiro. O resultado terá que ser sempre interpretado da mesma forma que os operandos (naturais ou inteiros). Como já foi referido anteriormente, quando se opera em código dos complementos, os operandos e o resultado estão definidos no mesmo número de bits.

Tomemos como exemplo as seguintes operações entre operandos de quatro bits:

Adição



Subtracção



Para o número de bits estabelecido, os operandos e o resultado, quando considerados números naturais, codificam números entre 0 e 15, quando considerados inteiros com sinal entre codificam números entre -8 e +7, ou seja, uma mesma combinação de bits representa um número natural e um número inteiro.

4.7.2 *Overflow*

Quando somamos ou subtraímos operandos naturais ou inteiros de n bits, o resultado pode exceder a capacidade de representação, ou seja, se os operandos forem de n bits o resultado poderá necessitar n+1 bits para poder ser representado o que constitui um problema.

Embora as unidades aritméticas presentes nos computadores que operam sobre n bits, disponibilizem normalmente o resultado em n bits mais o bit de arrasto, esta situação constitui um problema, pois os operandos e os resultados no computador são armazenados em registos de memória de n bits, o que implica que não possamos armazenar num único registo um resultado com n+1 bits. Por outro lado tornava complexa a utilização desse resultado como operando de posteriores operações.

Por esta razão é necessário que a unidade aritmética assinale o facto de se estar a exceder a capacidade de representação, pois de outro modo tomaríamos resultados errados como correctos. Quando os operandos são considerados números naturais o arrasto (*carry* ou *borrow* final), assinala este facto. No caso de os operandos serem inteiros com sinal é necessário adicionar um circuito que realize esta função. Nos inteiros com sinal, esta informação é normalmente denominada por OV (*overflow*).

Na Figura 4-17 estão representadas algumas operações de adição e subtracção produzindo erro por excesso e por defeito.

No conjunto $\mathbb{Z}$ existe excesso quando ao adicionarmos dois números positivos obtemos como resultado um número negativo, ou ao adicionarmos dois números negativos obtemos como resultado um número positivo.

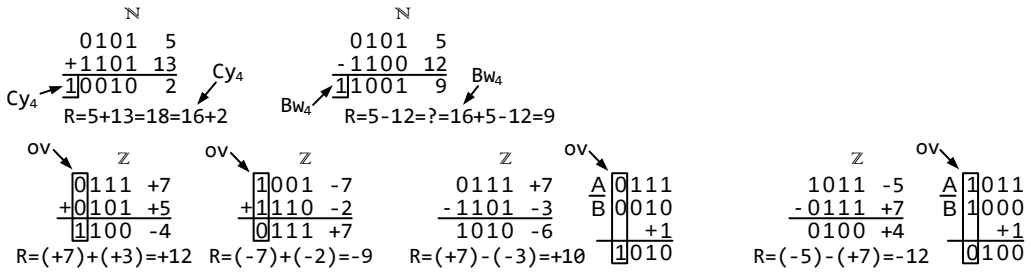


Figura 4-17

Existem várias formas de produzir informação de que existe OV, cada uma destas formas deve levar em conta a tecnologia que estivermos a utilizar na implementação e dos sinais que tivermos disponíveis.

Se estivermos a utilizar um módulo somador idêntico ao anteriormente apresentado e lógica discreta, uma primeira forma pode ser obtida da definição anteriormente descrita:

$$OV = \overline{A}_3.\overline{B}_3.S_3 + A_3.B_3.\overline{S}_3$$

Para esta implementação seriam necessários dois ICs, um de NANDs e outro de NORs como mostra a Figura 4-18.

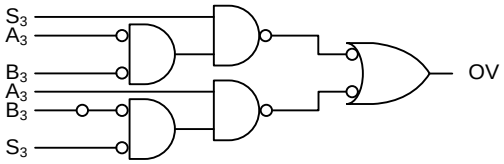


Figura 4-18

Como se pode observar na Figura 4-19 a implementação mais simples será $OV = Cy_n \oplus Cy_{n+1}$, pois existe *overflow* quando Cy_n é diferente de Cy_{n+1}.

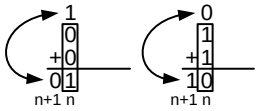


Figura 4-19

No entanto, se não dispusermos do arrasto Cy_n como acontece quando utilizamos um módulo IC somador completo, uma segunda forma mais simples que a primeira, utilizando um único IC de XORs, pode ser obtida da seguinte forma: dado que $S_n = A_n \oplus B_n \oplus Cy_n$ então pela propriedade do XOR que diz se $A = B \oplus C$ então $B = A \oplus C$ podemos obter para Cy_n a seguinte expressão:

$$Cy_n = A_n \oplus B_n \oplus S_n$$

Assim sendo, dado que $OV = Cy_n \oplus Cy_{n+1}$, podemos usar para *overflow* a seguinte expressão:

$$OV = (A_n \oplus B_n \oplus S_n) \oplus Cy_{n+1}$$

Que corresponde ao esquema mostrado na Figura 4-20.

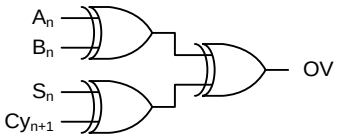


Figura 4-20

Dado que a subtracção é realizada utilizando a adição de A com o simétrico de B, então o mesmo circuito detector de *overflow* é válido para as operações de adição e subtracção.

4.8 Multiplicação Binária

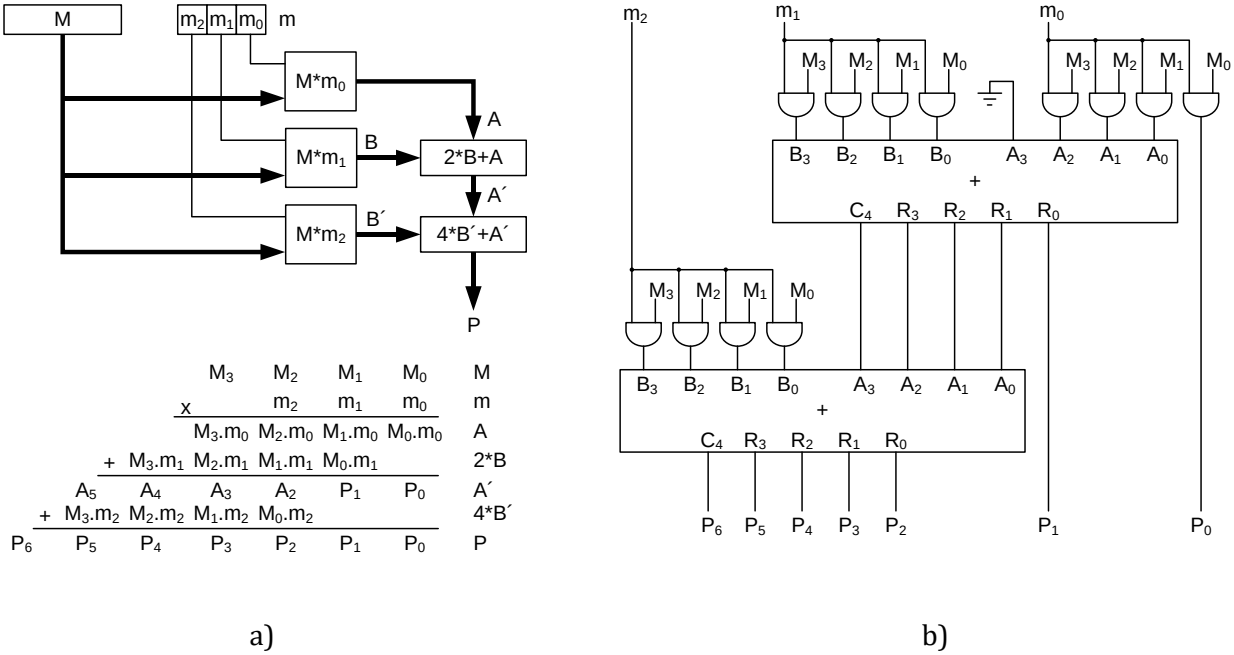
Outro exemplo de uma solução iterativa é a implementação do multiplicador *add-shift*.

Podemos construir um circuito capaz de realizar uma multiplicação utilizando o mesmo algoritmo que utilizamos quando multiplicamos dois números decimais e que é conhecido pelo algoritmo de *add-shift* (somar-deslocar), ou seja multiplicamos cada um dos dígitos do multiplicador por todos os dígitos do multiplicando e realiza-se a soma com as parcelas anteriores deslocado para a esquerda de um dígito. O produto **P** resultante da multiplicação de **M** (multiplicando) por **m** (multiplicador) é dado pela seguinte expressão:

$$P = M\,.m\; = \; M.(m_0 + 2.m_1 + \; 4.m_2 + \cdots + 2^{n-1}.m_{n-1})$$

$$P = M.m = M.m_0 + 2.M.m_1 + \cdots + 2^{n-1}.M.m_{n-1}$$

Multiplicar um número por uma potência inteira da sua base corresponde a deslocar para a esquerda esse número tantas vezes quanto o expoente da potência. Por exemplo $25 * 100 = 25 * 10^{\mathbf{2}}$ o que corresponde a deslocar o número 25 duas vezes para a esquerda obtendo-se 2500. Assim sendo podemos obter para a implementação da multiplicação de dois números binários (M com quatro bits e m com três bits) o diagrama de blocos da Figura 4-21 a).



a) b)

Figura 4-21

Como a tabuada da multiplicação na base dois corresponde ao AND lógico, obtemos para circuito do multiplicador de $M * m$ o esquema da Figura 4-21 b). Como podemos observar o módulo que realiza a operação $M * m_i$ corresponde a quatro portas AND entre m_i e cada um dos $M_{(0-3)}$ cujo padrão se repete. As multiplicações por 2 e por 4 correspondem a deslocar (*shift*) as ligações na entrada dos somadores.

4.9 Divisão Binária

A divisão realizada pelo algoritmo *Sub-Shift* tem uma solução idêntica à multiplicação por *add-shift*, como mostra a Figura 4-22 a). O divisor depois de multiplicado por um ou por zero, é subtraído ao dividendo; de seguida baixa-se o próximo algarismo do dividendo e que corresponde a deslocar (*shift*) para a esquerda o dividendo. Esta sequência é realizada sucessivamente até que não exista mais nenhum dígito do dividendo para baixar.

Aquando da subtracção, se for produzido *borrow* de peso n, porque o minuendo é menor que o subtraendo, o resultado da subtracção não é utilizado e em vez disso é propagado como resultado da operação o minuendo, o que corresponde à multiplicação do divisor por zero e subsequente subtracção do dividendo por zero. Quando esta situação ocorre é inserido o dígito 0 no quociente. Caso o minuendo seja maior ou igual ao subtraendo, são propagados os bits do resultado da subtracção e inserido o dígito 1 no quociente.

Como se pode observar no exemplo da Figura 4-22 a), quando se realiza subtracção o resultado é sempre inferior a 4 ou seja representável no mesmo número de bits. No entanto, se não for possível realizar a subtracção (por existir *borrow*) e o dígito de maior peso do minuendo tiver o valor 1, ao realizar o *shift* para esquerda do dividendo, o número assim gerado tem quatro bits, o que implica que embora o divisor tenha três bits a subtracção se realize a quatro bits.

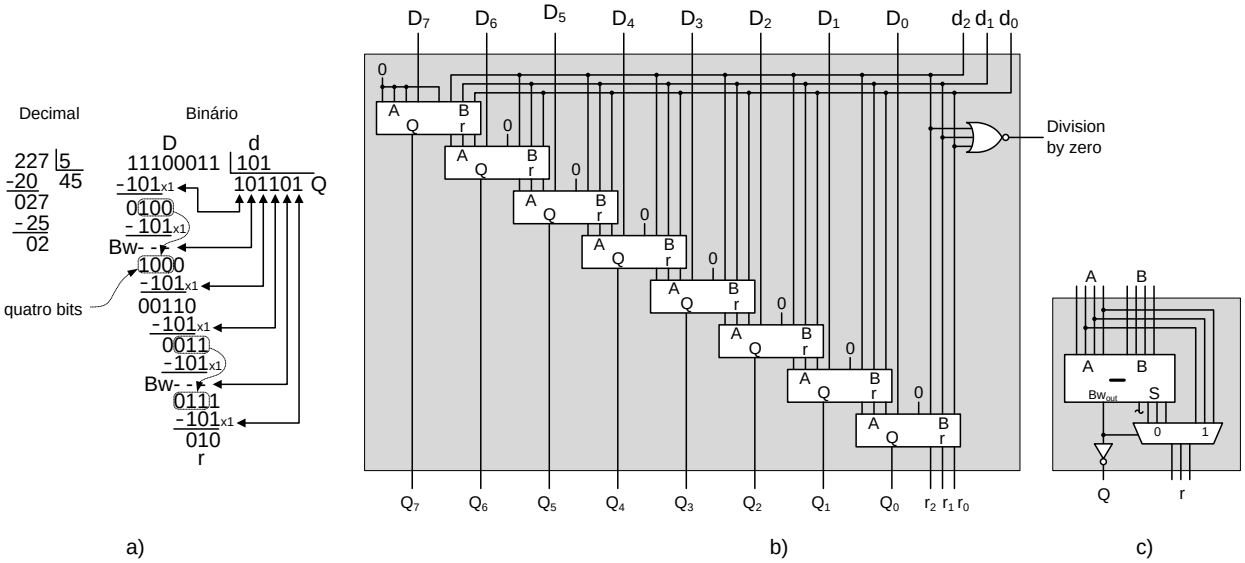


Figura 4-22

Na Figura 4-22 b) podemos ver a arquitectura de uma estrutura combinatória que implementa este algoritmo utilizando de forma recorrente um mesmo módulo a que denominaremos por módulo sub-pass. Na Figura 4-22 c) está representado o diagrama de blocos do módulo sub-pass e que corresponde a um subtractor do dividendo pelo divisor e um multiplexe que, função de existir ou não *borrow* (Bw_{out}), deixa passar para a saída o resultado da subtracção ou o minuendo. Em cada estágio, o quociente Q_i corresponde ao complementar do bit Bw_{out} e a saída do multiplexe corresponde ao resto $r_{0\text{-}2}$ da divisão.

Ao módulo que produz Q_7, são acrescentados três zeros à esquerda de D_7, pois o valor do divisor pode ser inferior a 4 (d2 igual a zero) ou inferior a 2 (d2 e d1 iguais a zero). Por exemplo, se o divisor valer 1, resulta num quociente igual ao dividendo.

4.10 Comparadores

O circuito comparador de números de n dígitos também recorre a uma solução iterativa. A relação de grandeza entre dois números pode ser obtida de duas formas: ou por subtracção e avaliação do resultado, ou por comparação dígito a dígito no sentido do maior para o menor peso. A segunda solução apresenta como vantagem a utilização de menos portas lógicas. A estrutura deste módulo ilustrada na Figura 4-23, traduz o facto de a comparação entre dois números se iniciar pelos algarismos de maior peso.

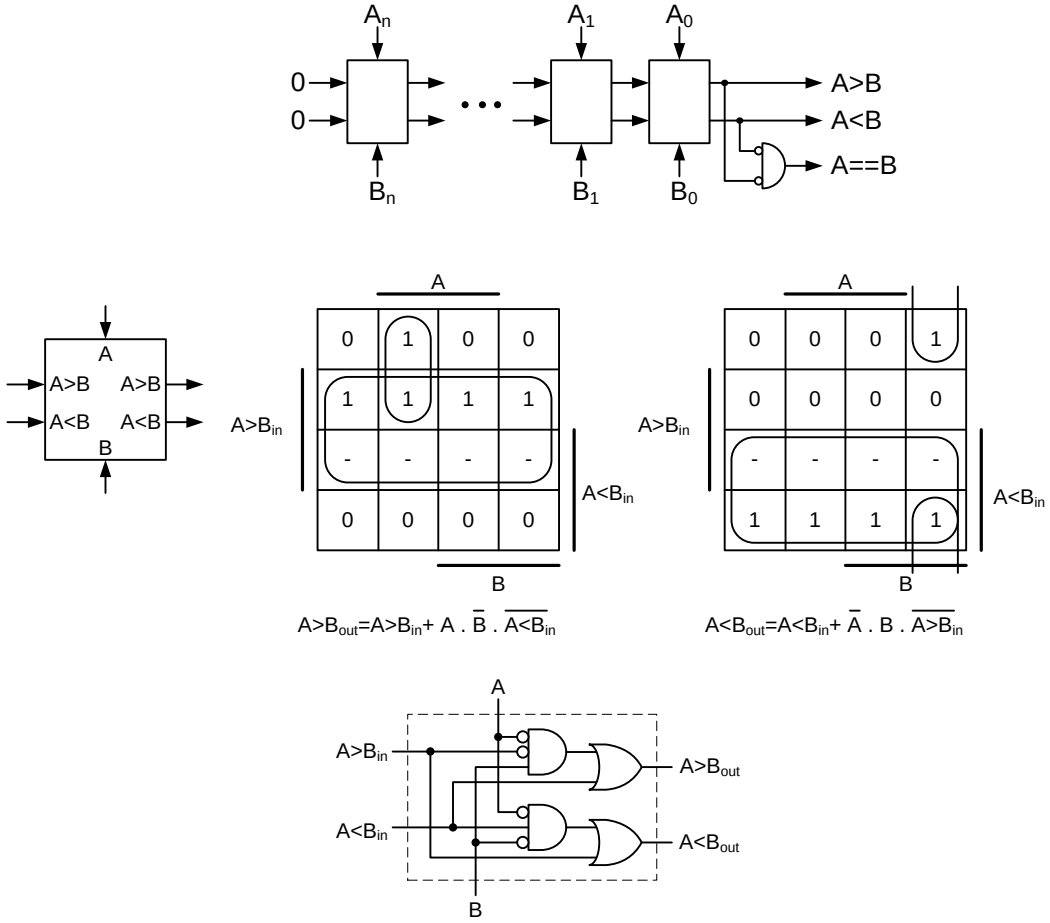


Figura 4-23

Se pretendermos utilizar este dispositivo para comparar números em código dos complementos, é necessário trocar ou inverter os bits de maior peso do operando A e B. A operação de complementação dos bits de maior peso é a mais utilizada nos processadores, pois a unidade aritmética disponível no processador pode não disponibilizar informação de *overflow*, tornando a determinação de qual o maior de dois números inteiros uma operação complexa. Se complementar o bit de maior peso de cada um dos operandos a informação de relação entre os dois inteiros é dada pelo bit de *borrow*.

4.11 *Barrel Shifter*

Na generalidade das linguagens de programação está disponível a operação *shift* (deslocamento) que corresponde a deslocar para a direita ou para a esquerda n vezes todos os bits de um operando. Esta operação é de extrema importância pois corresponde a multiplicar ou dividir por uma potência inteira de dois, dai estar disponível como operação nativa nas unidades de processamento. Para que esta operação seja eficiente em tempo de execução, a maioria dos processadores modernos apresentam uma implementação combinatória denominada por *Barrel Shifter*.

A Figura 4-24 apresenta a implementação de dois *barrel shifters* de quatro bits (D_{0-3}), um para a esquerda e outro para a direita. As duas linhas (S_{0-1}) estabelecem o número de posições (0 a 3) que o valor de entrada (D_{0-3}) é deslocado para a saída (Y_{0-3}). O bit Sin (*Serial in*) determina qual o valor lógico de entrada a ser inserido nos bits que vão sendo criados à direita ou à esquerda na saída Y_{0-3}.

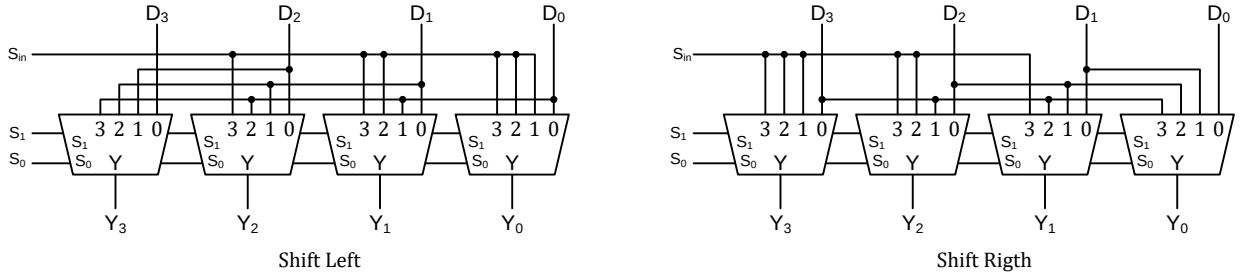


Figura 4-24

Outra operação normalmente disponibilizada nas unidades de processamento é a operação de rotação cuja implementação em *barrel* é apresentada na Figura 4-25.

A estrutura da Figura 4-25 realiza a rotação de todos os bits. Com esta arquitectura a rotação para a esquerda de três posições, corresponde a uma rotação para a direita de uma posição, ou seja, num *barrel shifter* de 2^n bits, rodar para a esquerda i posições, corresponde a rodar para a direita 2^n –i. Os bits que vão saindo por um extremo vão sendo inseridos pelo outro extremo.

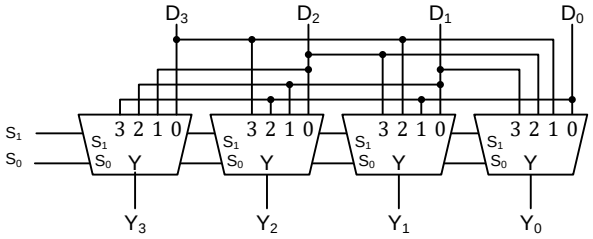


Figura 4-25

4.12 Códigos que não constituem um sistema de numeração

Como já foi anteriormente referido, uma sequência de bits pode representar diferentes coisas dependendo do código que estivermos a utilizar. Existem aplicações onde a codificação em números binários não é a mais conveniente, razão pela qual existirem várias codificações não numéricas. Entre os vários códigos distinguem-se, o BCD o *Gray* e o ASCII.

4.12.1 Código BCD

O sistema de numeração binária é o mais natural para um sistema digital, mas para um humano o mais natural é o decimal. Devido à complexidade, que um sistema digital capaz de converter um número de n dígitos decimais num número em base 2 apresenta, foi criado um código denominado BCD (*Binary Code Decimal*) que consiste em representar cada um dos algarismos decimais (0-9) na sua correspondente representação em base 2. Um sistema utilizando esta codificação converte para quatro bits um estímulo físico (uma tecla, um *thumb wheel switch*, etc.), opera estes bits e volta a converter o resultado da operação numa representação decimal. Para realizar operações aritméticas neste código têm que ser criadas estruturas que operem neste código, pois é necessário compensar a falta das seis configurações entre 10 a 15 (2^4=16). Cada conjunto de quatro bits constitui um algarismo do sistema, pelo que, por exemplo, uma soma será realizada da seguinte forma:

							1												
	8	6	4		1	0	0	0		0	1	1	0		0	1	0	0
+	9	7	5	+	1	0	0	1	+	0	1	1	1	+	0	1	0	1
1	8	3	9	1	0	0	1	0		1	1	0	1		1	0	0	1
					+	1	1	0		+	1	1	0					
				1	1	0	0	0	1	0	0	1	1					
				1		8				3						9		

O algoritmo utilizado na soma de dois números BCD é o seguinte:
É adicionado dígito a dígito BCD e é adicionado o valor 6 ao resultado, sempre que o resultado da soma dos dois dígitos BCD ultrapassa o valor 9 ou produz arrasto.
Com o aparecimento dos computadores este código caiu em desuso, devido à capacidade de cálculo disponível no processador, sendo preferível converter o valor decimal para base 2 operar em base 2 e voltar a converter o resultado para decimal.

4.12.2 Código *Gray*

Este código tem como característica, a mudança de um único bit entre configurações consecutivas. Este factor é extremamente importante quando se pretende informar através de vários bits e de forma simultânea a posição física de um determinado elemento móvel. Se a codificação da posição for em binário, por exemplo ao passarmos de 0111 (7) para 1000 (8) temos a modificação de 4 bits em simultâneo. Tratando-se da transmissão de informação de um sistema electromecânico para um sistema digital esta simultaneidade dificilmente seria garantida dada a velocidade de reacção dos sistemas digitais, produzindo-se desta forma um erro de posicionamento. Outra utilização importante deste código, são os contadores digitais construídos em tecnologia CMOS de baixo consumo, uma vez que nesta família o consumo de energia está directamente associado à alteração do estado de um transístor (bit). Com a utilização deste código produz-se uma distribuição uniforme do consumo e diminui-se o ruído na alimentação do circuito. A construção deste código é feita de forma espelhada como se mostra na Tabela 4-5 para uma codificação *gray* a três bits.

	S2	S1	S0
0	0	0	0
1	0	0	1
2	0	1	1
3	0	1	0
4	1	1	0
5	1	1	1
6	1	0	1
7	1	0	0

Tabela 4-5

4.12.3 Códigos Alfanuméricos

Muitas das aplicações de um computador não manipulam somente números, algumas manipulam textos compostos de números, letras e pontuação. A representação de texto num mostrador alfanumérico (CRT ,LCD, etc.) requer uma codificação que permita estabelecer um código para cada um dos símbolos representáveis. Existem vários códigos com este objectivo, como sejam, o EBCDIC, ASCII, Unicode, etc. Um dos códigos mais utilizado é o código ASCII (*American Standard Code Information Interchange*) que utiliza sete bits para codificar os vários caracteres e mais um de paridade para detecção de erro.

ASCII

Este código permite representar os algarismos de 0 a 9, as 26 letras do alfabeto (maiúsculas e minúsculas), caracteres especiais como $, #, %, etc., bem como 34 caracteres de controlo como sejam a mudança de linha, fim de linha etc. O bit de paridade corresponde ao bit de peso 7 e é resultado da operação XOR de todos os bits. Esta paridade é normalmente denominada paridade horizontal. Este bit é utilizado para detectar erros de comunicação quando se transfere texto entre dois sistemas.

4.13 ALU (*Arithmetic Logic Unit*)

Quando numa linguagem de alto nível (C, Java, etc..) especificamos uma expressão aritmética ou uma expressão lógica, subentende-se que o processador que vai executar o programa dispõe de uma unidade capaz de realizar tais operações. Esta unidade, denominada por ALU (*Arithmetic Logic Unit*), é responsável por executar as operações básicas da aritmética e da lógica.
Admita que pretende realizar uma ALU, que executa as seguintes operações:

$$R = A + B + CyBw_{in}$$
$$R = A - B - CyBw_{in}$$
$$R = A + 1$$
$$R = A - 1$$
$$R = A \,\&\, B$$
$$R = A \mid B$$

Na Figura 4-26 está representado o diagrama da ALU evidenciando as várias entradas e saídas.

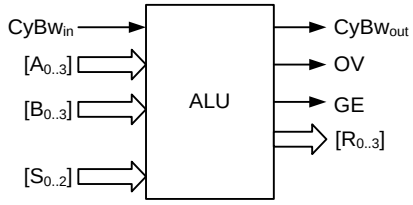


Figura 4-26

$[A_{0..3}]$ e $[B_{0..3}]$ são dois operandos de 4 bits, $CyBw_{in}$ é um bit de peso zero e que é tomado como *carry* na soma e *borrow* na subtracção. A existência desta entrada vai permitir aos programas executarem operações de soma ou subtracção sobre operandos com um número de bits maior que os suportados pela ALU. Esta soma, dita realizada em série, consiste em dividir os operandos em conjuntos de n (dimensão da ALU) bits, e tomar cada um dos conjuntos como um dígito. Por cada operação que realiza sobre o conjunto de n bits, memoriza o resultado R e o arrasto $CyBw_{out}$ da operação e submete o arrasto assim memorizado, como bit $CyBw_{in}$ da operação sobre o próximo conjunto de n bits.
Como a ALU realiza uma de seis operações, é necessário que existam sinais de entrada que determinem qual a operação pretendida. Por serem seis operações poderemos codificar com três bits $[S_{0..2}]$ cada uma das operações como mostra a Tabela 4-6.

S_2	S_1	S_0	Operação	$CyBw_{out}$	OV	GE
0	0	0	ADDC	•	•	-
0	0	1	SUBB	•	•	•
0	1	0	INC	•	•	-
0	1	1	DEC	•	•	-
1	0	0	AND	-	-	-
1	0	1	OR	-	-	-

Tabela 4-6

Para além do resultado R, a ALU põe disponíveis dois indicadores de excesso: $CyBw_{out}$ e OV, e um indicador relacional GE (*Greater or Equal*). Estes indicadores são normalmente denominados por *flags*.

4.13.1 Funções combinatórias

Dada a complexidade do exercício proposto, a sua solução não é passível de uma abordagem como a que temos feito até aqui, através de uma tabela de verdade e portas lógicas. Como já aconteceu anteriormente, foi necessário passar do nível do transístor para o nível da porta lógica, sendo necessário agora passar novamente para um nível superior de descrição, dividindo o problema em blocos funcionais numa lógica de dividir para reinar, identificando as entradas, saídas e funcionalidade de cada bloco. Esta divisão irá sendo feita de forma hierárquica, até que cada bloco apresente uma complexidade de implementação aceitável. Esta abordagem apresenta como grande vantagem a possibilidade de reestruturação, optimização e teste localizada.

Comecemos por dividir a estrutura em dois grandes blocos: o aritmético e o lógico. Utilizaremos o bit S_2 para os distinguir (seleccionar) como mostra a Figura 4-27.

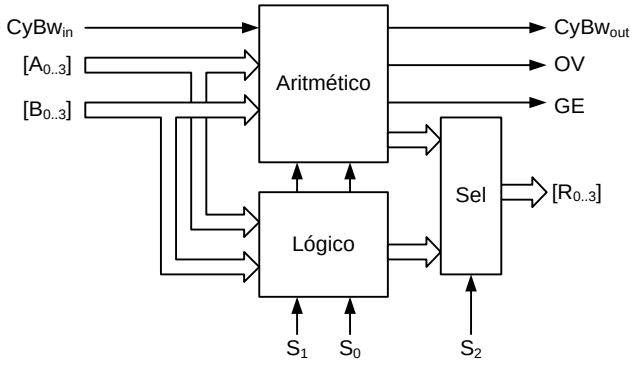


Figura 4-27

4.13.2 Bloco Aritmético

O bloco aritmético terá como elemento principal um somador de dois números de 4 bits. Este elemento deverá ser reutilizado sempre que possível para as várias operações por ser consumidor de um grande número de lógica. O bloco aritmético realiza as seguintes operações:

$$A + B + CyBw_{in}$$
$$C_0 = CyBw_{in}$$

$$A - B - CyBw_{in} = A + (-B) - CyBw_{in} = A + \overline{B} + \underbrace{1 - CyBw_{in}}_{C_0} = A + \overline{B} + \overline{CyBw}_{in}$$

$$C_0 = 1 - CyBw_{in} = \overline{CyBw_{in}}$$

$$A + 1 = A + 0 + 1$$
$$B = 0 \; e \; C_0 = 1$$

$$A - 1 = A + (-1) + 0$$
$$B = -1 \; e \; C_0 = 0$$

A Figura 4-28 apresenta uma possível solução para o bloco aritmético.

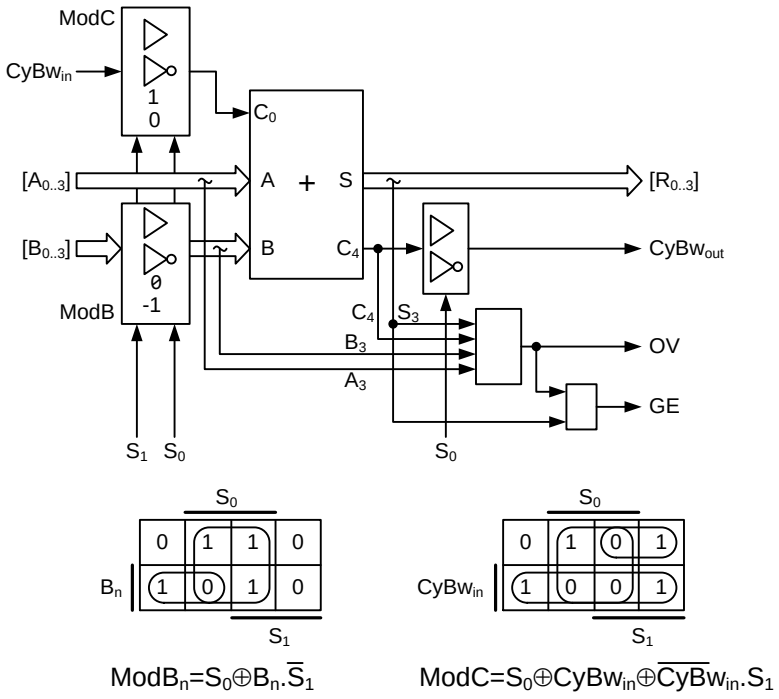


Figura 4-28

A implementação dos indicadores relacionais não recorre a comparadores. Estes indicadores são implementados à custa da observação do resultado da subtracção, assim sendo, estes indicadores só têm validade quando a ALU está a realizar uma subtracção entre A e B com $CyBw_{in}=0$.

As siglas normalmente utilizadas nos operadores relacionais, seguem as estabelecidas pela Intel na arquitectura IA32, e são as seguintes:

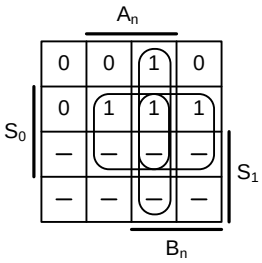
- A (*Above*) A é maior que B considerando A e B pertencente ao conjunto$\mathbb{N}$.
- AE (*Above or Equal*) A é maior ou igual a B considerando A e B pertencente ao conjunto $\mathbb{N}$.
- B (*Below*) A é menor que B considerando A e B pertencente ao conjunto $\mathbb{N}$.
- BE (*Below or Equal*) A é menor ou igual a B considerando A e B pertencente ao conjunto $\mathbb{N}$.
- G (*Greater*) A é maior que B considerando A e B pertencente ao conjunto$\mathbb{Z}$.
- GE (*Greater or Equal*) A é maior ou igual a B considerando A e B pertencente ao conjunto $\mathbb{Z}$.
- L (*Less*) A é menor que B considerando A e B pertencente ao conjunto $\mathbb{Z}$.
- LE (*Less or Equal*) A é menor ou igual a B considerando A e B pertencente ao conjunto $\mathbb{Z}$.

A existência de *borrow* indica que A é menor que B entendidos estes como números naturais, ou seja, o *borrow* indica que não é possível retirar o valor B a A.

Um resultado positivo indica que A é maior ou igual a B entendidos estes como inteiros com sinal. O indicador GE não pode depender exclusivamente do bit de sinal do resultado, pois a operação que está a ser realizada pode estar a exceder a representação, razão pela qual a implementação deste indicador terá que tomar em consideração o sinal OV.

4.13.3 Bloco Lógico

Quanto ao bloco lógico, poderemos extrair do mapa de *Karnaugh* a função de cada um dos bits de saída como mostra a Figura 4-29.



$Logic_n = A_n.B_n + S_0.A_n + S_0.B_n$

Figura 4-29

4.13.4 Implementação

A arquitectura a sintetizar não pode ser completamente indissociável da tecnologia que vamos utilizar na implementação. O projectista deverá conhecer as várias componentes e soluções tecnológicas disponíveis no mercado a fim de optar pela melhor solução.

Na Figura 4-30 é apresentada para cada um dos blocos aritmético e lógico, uma solução alternativa àquela que recorria exclusivamente a portas lógicas, e que se baseia na utilização de um módulo cuja funcionalidade é designada por multiplexer, podendo também ser referida como selector, por transferir para a saída uma das n entradas seleccionadas pelos bits S. O módulo multiplexer será estudado no capítulo 5.

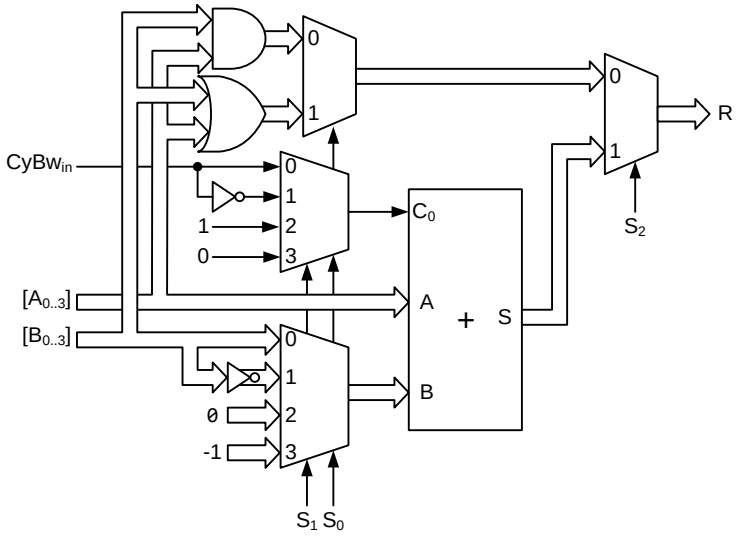


Figura 4-30

4.14 Exercícios do capítulo 4

[1] Considere uma ALU a quatro bits que realiza a operação R=A-1-CyBwi. Considerando os valores A=1001 e CyBwi=1 responda às seguintes alíneas:

a) indique a configuração que deverá estar presente em R;
b) justifique os valores das saídas CyBwOut (saída de carry/borrow) e Ov (overflow) nesse caso.

[2] Considere que atribuíam aos operandos da ALU (Unidade Aritmética e Lógica), os valores A=1111(2), B=0110(2) e CyBwi = 1. A operação seleccionada é R=A-B-CyBwi. Justifique os valores obtidos em R e nos indicadores CyBwOut, OV, GE e BL.

[3] Admita que numa ALU, foi seleccionada a operação R=A-B com A=0101 e B=1101, indique qual o valor presente em R e diga justificando quais os valores das quatro flags BL, GE, CyBwOut e Ov.

[4] Complete os campos das tabelas assumindo que numa ALU de 4 bits está seleccionada a operação R = A - B - CyBwi.

a)

		R	A	B	CyBwi	CyBwO	Ov	BL	GE
Base 2		1000							
Base 10	natural						–	1	–
	relativo					–		–	0

b)

		R	A	B	CyBwi	CyBwO	Ov	BL	GE
Base 2		1011	0101						
Base 10	natural				1		–	1	–
	relativo					–		–	1

c)

		R	A	B	CyBwi	CyBwO	Ov	BL	GE
Base 2		0001							
Base 10	natural				0		–	0	–
	relativo					–		–	0

[5] Realize as seguintes operações aritméticas. Para cada uma das operações, quais os valores de A, B e C que levam a que exista excesso, entendidos os operandos e o resultado no conjunto dos números naturais e nos inteiros com sinal.

	1	1	B
+	$\overline{A}$	A	0
	R2	R1	R0

a)

	$\overline{A}$	1	0
+	1	C	B
	R2	R1	R0

b)

	C	1	0
-	1	$\overline{A}$	C
	R2	R1	R0

c)

	0	1	C
-	A	C	B
	R2	R1	R0

d)

[6] Indique dois valores para A e B, tais que os indicadores relacionais apresentem os seguintes valores:

a) GE=1 e BL=0;
b) GE=0 e BL=0;
c) CyBwOut=1 e Ov=0;

[7] Num dado processo onde participam seis entidades E_0, E_1, E_2, E_3, E_4, e E_5, o gestor do processo pretende saber alguma informação sobre a actividade conjunta das várias entidades. Esta informação está disponível em quatro saídas como mostra a Figura 4-31 e cuja informação está descrita na Tabela 4-7.

N	Nenhuma
MM	Mais de Metade
T	Todas
I	Em número Impar

Tabela 4-7

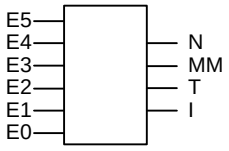


Figura 4-31

Soluções:

[1]

a) R=7.

b) CyBwOut=0, Ov=1.

[2]

R=8; CyBwOut=0, Ov=1, GE=0, BL=0.

[3]

R=8; CyBwOut=1, Ov=1, GE=1, BL=1.

[4]

a)

		R	A	B	CyBwi	CyBwO	Ov	BL	GE
Base 2		1000	0000	0111					
Base 10	natural	8	0	7	1	**1**	–	1	–
	relativo	-8	0	+7		–	**0**	–	0

b)

		R	A	B	CyBwi	CyBwO	Ov	BL	GE
Base 2		1011	0101	1010					
Base 10	natural	11	5	10	0	**1**	–	1	–
	relativo	-5	+5	-6		–	**1**	–	1

c)

		R	A	B	CyBwi	CyBwO	Ov	BL	GE
Base 2		0001	1000	0111					
Base 10	natural	1	8	7	0	**1**	–	0	–
	relativo	+1	-8	+7		–	**1**	–	0

[5]

a) R0=B, R1=$\overline{A}$, R2=0; CyBwOut=1, Ov=$\overline{A}$.

b) R0=B, R1=$\overline{C}$, R2=$C \oplus A$; CyBwOut=$\overline{A} + C$, Ov=$\overline{A}.\overline{C}$.

c) R0=C, R1= $C \oplus A$, R2=$\overline{A.C}$; CyBwOut=$\overline{A.C}$, Ov=$A + \overline{C}$.

d) R0= $B \oplus C$, R1= $B.\overline{C}$, R2=A; CyBwOut=A, Ov=A.

[6]

a) A=1111, B=1000;

b) A=1000, B=0111;

c) A=0001, B=0010;

[7]