UNIDADE ARITMÉTICA

1. Generalidades
2. Operações Aritméticas
   - 2.1. Operações em código binário
     - 2.1.1. Soma paralelo
     - 2.1.2. Soma série
     - 2.1.3. Somador/subtractor
   - 2.2. Operações em código BCD
     - 2.2.1. Soma paralelo
     - 2.2.2. Subtracção paralelo
   - 2.3. Somadores/subtractores utilitando contadores
   - 2.4. Multiplicação em código binário
     - 2.4.1. Multiplicador paralelo
     - 2.4.2. Multiplicador série
   - 2.5. Divisão em código binário
   - 2.6. Multiplicação e divisão utilitando contadores
     - 2.6.1. Multiplicação
     - 2.6.2. Divisão
   - 2.7. Outras operações aritméticas
3. Circuitos aritméticos integrados.

UNIDADE ARITMETICA

1. Generalidades

Abstraindo toda a natural complexidade dos circuitos de um computador, podemos assumir que o circuito base das operações aritméticas que o mesmo executa é o circuito adicionador completo ou "full-adder".

Este circuito, que adiciona dois bits (e a "carry" anterior) dando como resultado a soma e uma "carry", pode ser usado para executar subtracções, multiplicações, divisões ou até outras operações mais complexas desde que, determinada sequência de passos seja seguida. A esta sequencia de passos chama-se o algoritmo de execução da operação.

Os algoritmos podem ser definidos por software ou implementados em hardware, sendo no entanto em qualquer dos casos uma mesma essencia.

Para implementar em hardware determinados algoritmos, ter-se-á que rodear os full adders de circuitos lógicos que executem as manipulações de dados de tal modo que os passos que o constituem sejam seguidos.

Com esta opção, transferimos para hardware, à custa de uma maior complexidade dos circuitos, certas operações que poderiam contudo ser ordenadas por software. Como contrapartida obtemos uma maior rapidez de operação do que se os passos do algoritmo fossem definidos pelo programa.

2. Operações aritméticas.

As operações aritméticas são executadas na grande maioria dos computadores em código binário.

É contudo possível executar operações aritméticas em outros

códigos (BCD, excesso de 3, etc) usando como base os mesmos full adders desde que se executem os algoritmos apropriados.

As operações em BCD justificam-se em pequenas máquina onde a entrada dos dados é feita em decimal e após pequenas operações aritméticas ou logicas os dados devem apresentar-se numa saida, evidentemente em decimal. Dado que a codificação/descodificação decimal-BCD é bastante mais simples do que a binário-decimal, escolhe-se o código BCD para fazer as operações.

Quanto às vantagens que podem levar a trabalhar em código excesso de 3 elas são principalmente de simplificação de algoritmos. Referi-los-emos mais tarde.

No que se refere à implementação dos algoritmos das operações podemos seguir duas filosofias distintas:
- execução dos algoritmos em paralelo, ou seja simultâneamente, ou execução em série, ou seja, sequencialmente.
- Estes dois modos estão relacionados entre si pelas variáveis, rapidez de execução e complexidade do circuito, logo custo da máquina.

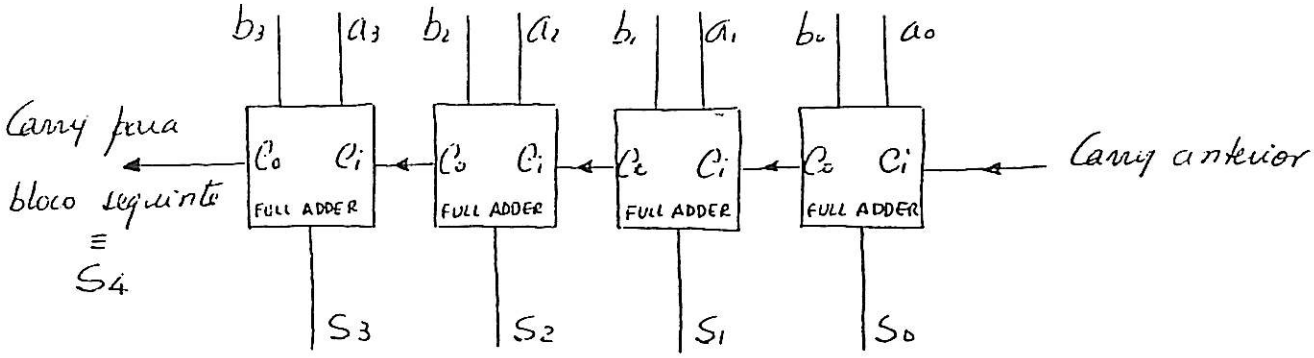
2.1 Operações em códigos binário

2.1.1 Soma paralelo

Dizemos que uma operação é efectuada de um modo paralelo quando dispomos num mesmo instante de todos os bits que constituem os dados a operar e os tratamos simultâneamente.

Isto implica que os circuitos operadores sejam tanto mais quanto maior (em comprimento de palavra) sejam os numeros a operar. Em contrapartida o tempo de execução da operação apenas é dependente do atraso introduzido pela propagação dos sinais ao longo dos circuitos utilizados.

Conforme referido anteriormente no capítulo de OPERAÇÕES ARITMÉTICAS, usando blocos adicionadores completos (full-adders) podemos implementar de um modo simples e imediato um circuito somador paralelo:



Circuito adicionador de 4 bits, paralelo, com propagação do sinal de transporte (em inglês: ripple carry)

Note-se que esta propagação do sinal de transporte (do F.A. somador dos bits menos significativos, para o F.A dos bits mais significativos) tem como consequência que o resultado presente à saída S0 ... S4, só será válido após ter decorrido o tempo necessário para que eventuais carrys que apareçam, tenham tempo para se propagar, e corrigir os resultados iniciais.

Também como já foi descrito no capítulo acima mencionado, a fim de conseguirmos maior rapidez na execução desta operação, podemos implementar um circuito semelhante ao apresentado, mas em que os diversos sinais de Ci são gerados por circuitos combinacionais, apenas a partir das variáveis de entrada (look ahead carry).

Conseguimos dispensar assim, o tempo de propagação do carry ao longo do circuito.

Para obtermos circuitos somadores paralelos para números com n bits de comprimento (p. ex. 16 bits) basta ligar em

cascata (isto é, ligar a saída Carry out à entrada Carry in do bloco seguinte) tantos circuitos como o descrito na página anterior quantos necessários.

Analisemos agora os dois circuitos referidos na pag. anterior sob o ponto de vista de tempo de execução da operação.

No 1º caso (circuito adicionador paralelo com ripple carry) quando o comprimento em bits dos números a adicionar aumenta, o tempo de propagação de carry pode tornar-se importante:

utilizando circuitos lógicos de tecnologia TTL, em que o tempo de atraso da entrada para a saída será da ordem dos 30 ns por cada Full adder, virá, para números com 16 bits de comprimento, p. ex. um tempo de execução da operação de cerca de 480 ns.

No 2º caso (circuito somador paralelo com "look ahead carry generator") o tempo de execução da operação é independente do comprimento dos números a operar e será da ordem de grandeza de 60 a 80 ns.

Note-se no entanto que a complexidade do circuito aumenta fortemente com o comprimento em bits dos números a somar

Como solução intermédia pode optar-se por blocos adicionadores de 4 bits com "look ahead carry generator" e ligar estes blocos entre si em cascata, ou seja, com propagação de sinal de carry.

Neste caso, e para os mesmos 16 bits de comprimento dos números a somar, o tempo necessário à operação andará pela ordem dos 100/120 ns

2.1.2 Soma em série

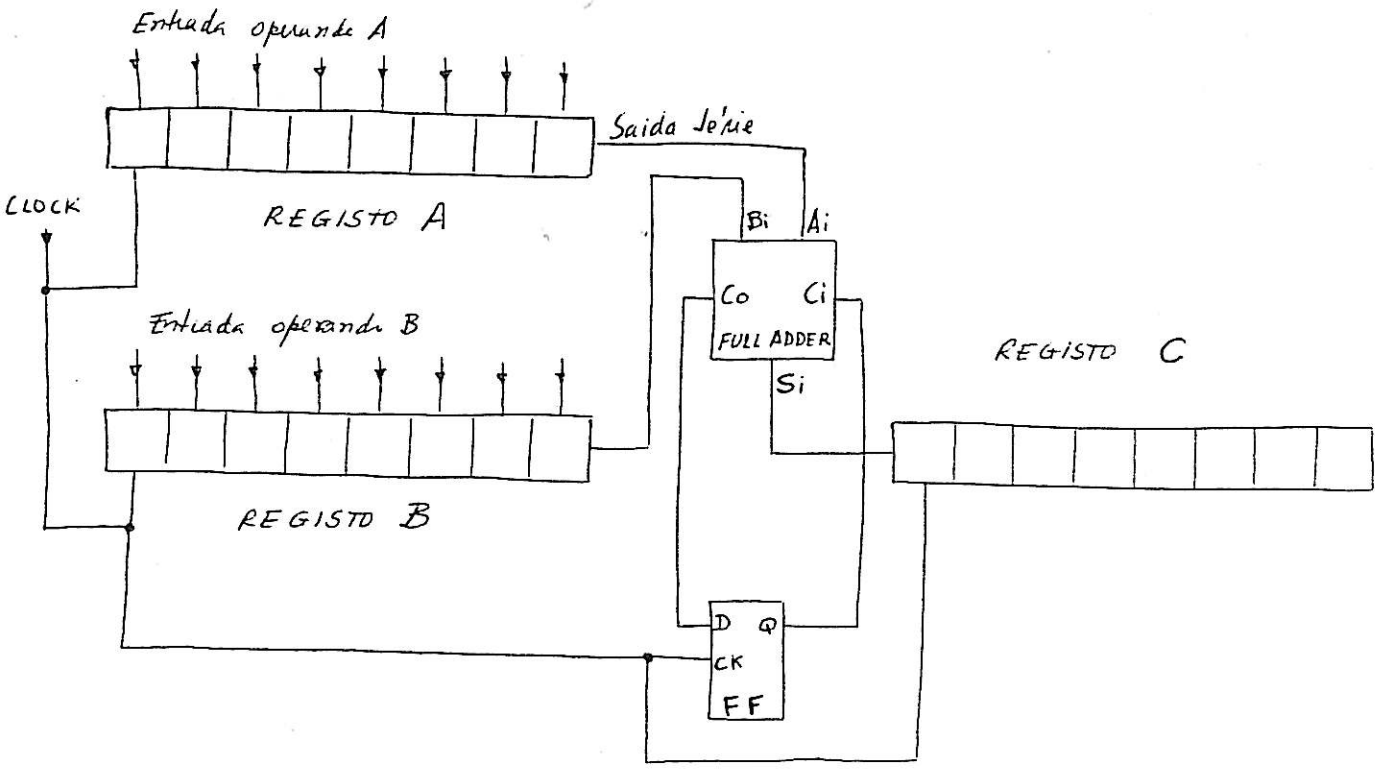
Dizemos que uma operação sobre dois números é executada de um modo série, quando a mesma incide sobre um par de bits de cada ret (um de cada operando) e começando pelos menos significativos.

Deste modo, à entrada do circuito operador vão aparecer de um modo sequencial no tempo, pares de bits dos dois números a operar.

Isto conduz-nos a que um só bloco operador de 2 bits, possa operar números de comprimento tão grande quanto se queira, sendo a única limitação existente, o tempo necessário para completar a mesma.

Vejamos como poderíamos implementar um circuito que realizasse a soma série de dois números, p. ex. de 8 bits de comprimento.

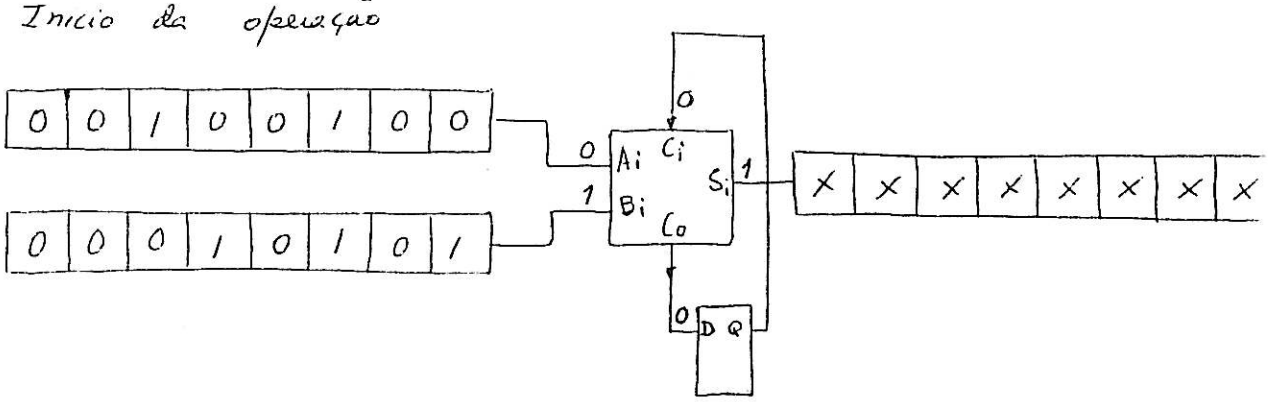
Para isso, dispomos de um único Full adder e de três (que poderão ser apenas dois, como se verá) registadores de deslocamento (shift-registers)



1- Façamos transferir para os registos A e B os dois números a somar, (bit menos significativo à direita) o que pode ser feito p. ex. através das entradas paralelo do shift-register.

Exemplo: A = 36 , B = 21

Início da operação



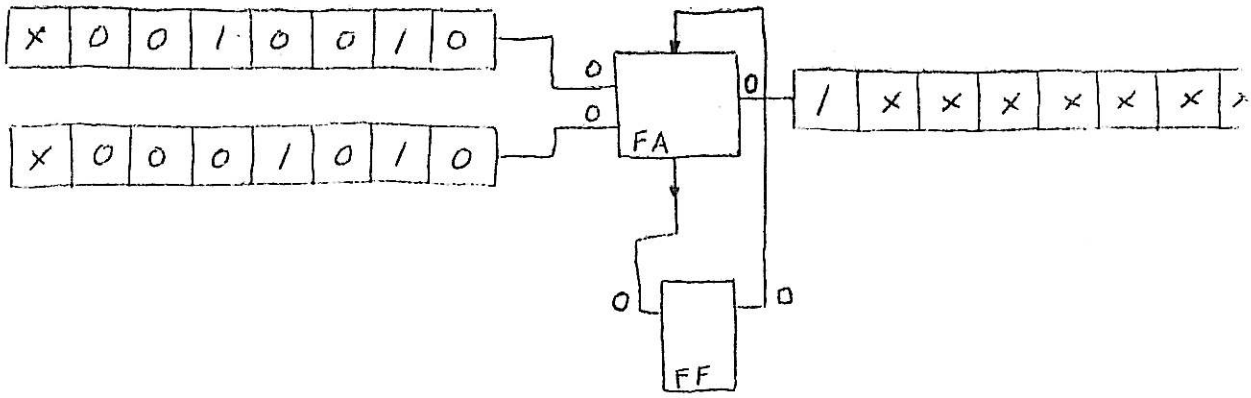
2- Comecemos a dar ordens de deslocamento para a direita aos shift registers, actuando a linha de clock, o que também vai efectuar a memorização da carry anterior no FF D.

Vejamos que à entrada do F.A. vão aparecendo sequencialmente os pares de bits a somar assim como a carry da soma anterior que entretanto foi armazenada no FF D.

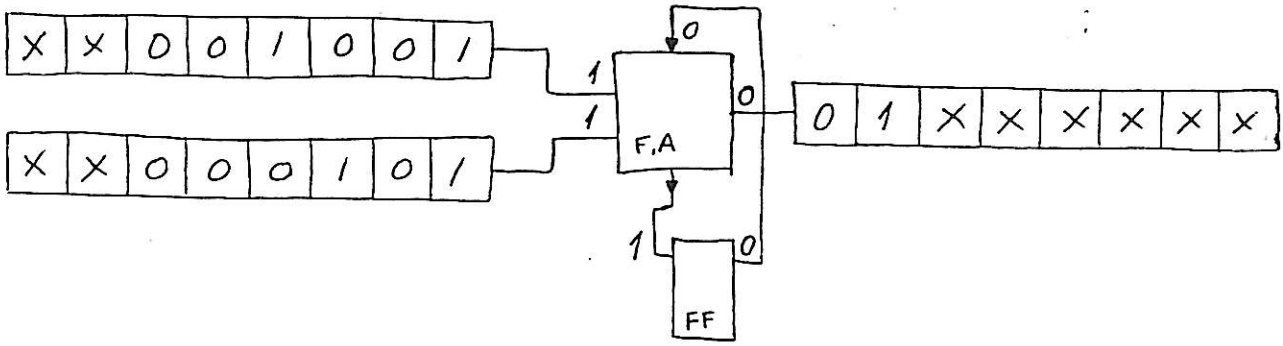
Os resultados parciais também vão sendo transferidos para o shift register C.

Vejamos:

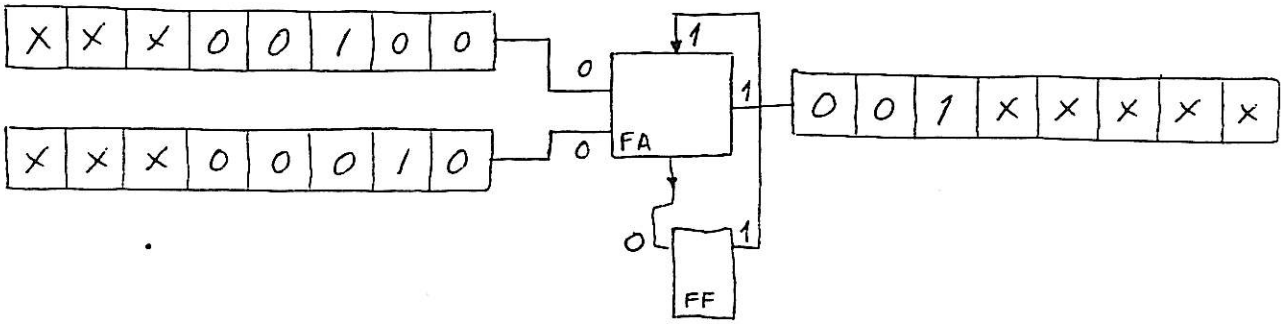
Após o 1º período do clock (⊓).



Após o 2º período do clock



Após o 3º período do clock



E assim sucessivamente até que, após o 8º sinal de clock, o resultado da operação ficará armazenado no Registo C.

Observações.

1º- O facto de após o 8º final de clock ficar um "um" no FF D significa, quando acontece, que a soma dos bits de peso 2^7 (8º bit) produziu uma carry, ou seja, a operação conduziu a um "overflow".

2º Se interessar que, contrariamente ao que sucedeu no exemplo anterior, algum dos operandos, ou ambos, não sejam destruídos no decorrer da operação (p. ex. para permitir várias somas com um mesmo nº fixo), bastará ligar entre si a saída e entrada série do shift register respetivo. Isto originará que esse operando circule dentro do shift e após o 8º impulso de clock ocupe de novo a posição inicial.

3º- Caso interesse também, o registo C pode ser dispensado, sendo nesse caso resultado armazenado num dos registos dos operandos.

Para tal, mais não é necessário do que ligar a saída do F.A. à entrada série do registo escolhido. Como o operando se vai deslocando uma posição para a direita por cada período do clock, vai deixando posições livres, à esquerda, posições essas que vão sendo ocupadas pelos bits do resultado que entretanto se está a formar.

Isto permite executar adições sucessivas, ou seja, acumular num registo a soma acumulada de vários números. Neste caso, o registo onde tal se efectua é denominado geralmente por - acumulador.

Para obter um somador série para números de comprimento igual a n bits, bastaria aumentar o comprimento dos registos, sendo a restante lógica a mesma.

Note-se no entanto, que o tempo de execução de uma soma de dois números aumenta com o comprimento em bits dos mesmos.

Por exemplo, para número com 16 bits de comprimento e considerando as frequências típicas para o sinal de clock da tecnologia TTL, da ordem dos 10 MHz, o tempo de execução de uma operação de soma com o circuito acima apresentado será da ordem dos 1600 ns = 1.6 μs (16 operações elementares de shift e soma × 100 ns / operação)

2.1.3. Somador - Subtractor

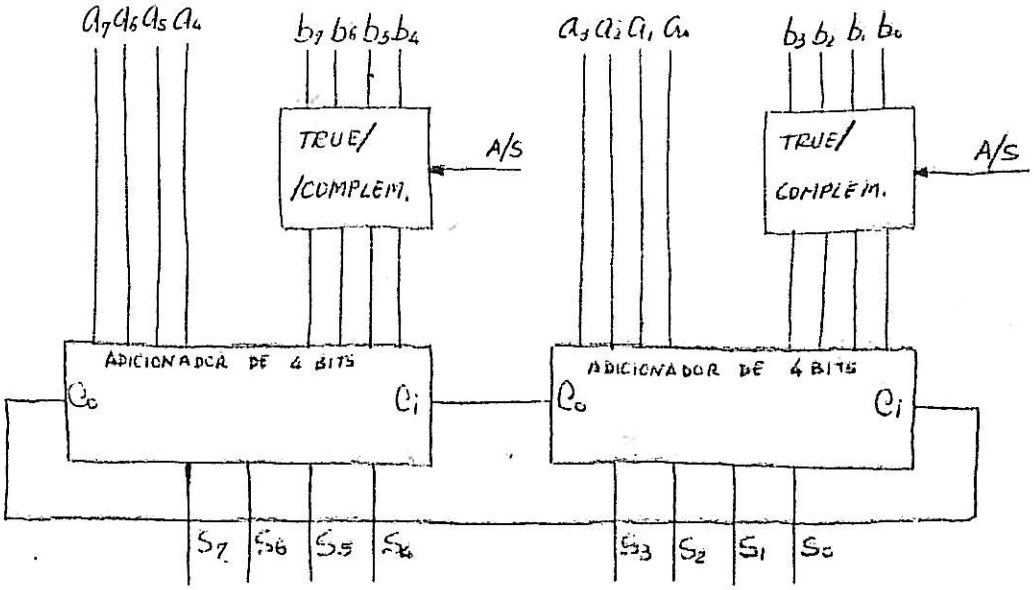
De acordo com o que foi referido no capitulo "OPERAÇÕES ARITMETICAS", para executarmos uma subtracção em código binário, não temos mais que complementar (a 1 ou a 2) o numero a subtrair e adicionarlo em seguida ao subtraendo.

Esta adição pode ser feita através de qualquer um dos métodos anteriores.

Note se ainda, que, no caso de trabalharmos em complemento de 1 para representar o subtractor, haverá que providenciar para que o sinal de transporte eventualmente gerado pelos bits mais significativos, (end-around-carry), seja somado ao bit menos significativo do resultado.

Exemplo:

Adicionador/subtractor, em complemento de 1, para numeros com 8 bits de comprimento (paralelo c/ "ripple carry")



Os blocos TRUE/COMPLEMENT (como são conhecidos na literatura anglo-saxonica) transferem os bits de entrada para a saida ou complementam-nos de acordo com a informação presente na entrada de controlo A/S.

Portanto e mediante a actuação deste sinal de comando o bloco apresentado pode efectuar as operações

$A + B$ ou $A - B$

Note-se a ligação de C_o do 2º adicionador com C_i do 1º, o que vai efectivar a soma do end-around-carry, quando existir, ao bit menos significativo do resultado.

Note-se também que este circuito permite a soma e subtracção de 8 bits, dos quais no entanto um (o mais significativo) será o bit de sinal.

O facto de trabalharmos em complementos de 1, com a consequência imediata de se ter que somar a end-around-carry ao bit menos significativo, vai originar que o resultado da operação apenas seja valido após a propagação do end-around-carry do bit menos significativo para o mais significativo. Logo, este circuito como somador é mais rápido do que como subtractor.

Para realizar uma subtracção por um método série, poderíamos usar o circuito somador série descrito atrás, providenciando apenas para que, aquando da transferência paralelo dos numeros, para os registos A e B, o numero a subtrair seja complementado a 1.

Isto conseguir-se-ia com um circuito TRUE/COMPLEMENT intercalado nas entradas paralelas do registo B (p.ex) e que comandaria a execução de uma soma ou subtracção.

Como no entanto, e depois de efectuar a soma, teríamos que somar a end-around-carry ao resultado, isso torna este método bastante complicado.

Para obviar a este inconveniente, poderíamos trabalhar em complementos de 2, o que, embora complicando o circuito complementador, não necessitaria desta operação final.

códigos (BCD, excesso de 3, etc) usando como base os mesmos full adders desde que se executem os algoritmos apropriados.

As operações em BCD justificam-se em pequenas máquinas onde a entrada dos dados é feita em decimal e após pequenas operações aritméticas ou logicas os dados devem apresentar-se numa saida, evidentemente em decimal. Dado que a codificação/descodificação decimal-BCD é bastante mais simples do que a binário-decimal, escolhe-se o código BCD para fazer as operações.

Quanto às vantagens que podem levar a trabalhar em código excesso de 3 elas são principalmente de simplificação de algoritmos. Referi-los-emos mais tarde.

No que se refere à implementação dos algoritmos das operações podemos seguir duas filosofias distintas:
- execução dos algoritmos em paralelo, ou seja simultâneamente, ou execução em série, ou seja, sequencialmente.

Estes dois modos estão relacionados entre si pelas variáveis, rapidez de execução e complexidade do circuito, logo custo da máquina.

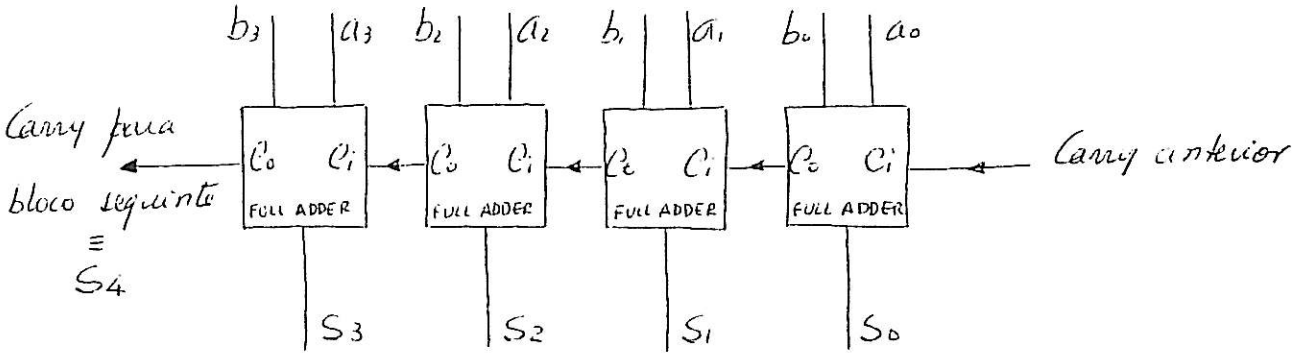
2.1 Operações em códigos binário

2.1.1 Soma paralelo

Dizemos que uma operação é efectuada de um modo paralelo quando dispomos num mesmo instante de todos os bits que constituem os dados a operar e os tratamos simultâneamente.

Isto implica que os circuitos operadores sejam tanto mais quanto maior (em comprimento de palavra) sejam os numeros a operar. Em contrapartida o tempo de execução da operação apenas é dependente do atraso introduzido pela propagação dos sinais ao longo dos circuitos utilizados.

Conforme referido anteriormente no capítulo de OPERAÇÕES ARITMÉTICAS, usando blocos adicionadores completos (full-adders) podemos implementar de um modo simples e imediato um circuito somador paralelo:



Circuito adicionador de 4 bits, paralelo, com propagação do sinal de transporte (em inglês: ripple carry)

Nota-se que esta propagação do sinal de transporte (do F.A. somador dos bits menos significativos, para o F.A dos bits mais significativos) tem como consequência que o resultado presente à saída S0 ... S4, só será válido após ter decorrido o tempo necessário para que eventuais carrys que apareçam, tenham tempo para se propagar, e corrigir os resultados iniciais.

Também como já foi descrito no capítulo acima mencionado, a fim de conseguirmos maior rapidez na execução desta operação, podemos implementar um circuito semelhante ao apresentado, mas em que os diversos sinais de Ci são gerados por circuitos combinacionais, apenas a partir das variáveis de entrada (look ahead carry).

Conseguimos dispensar assim, o tempo de propagação do carry ao longo do circuito.

Para obtermos circuitos somadores paralelos para números com n bits de comprimento (p. ex. 16 bits) basta ligar em

cascata (isto é, ligar a saída Carry out à entrada Carry in do bloco seguinte) tantos circuitos como o descrito na página anterior quantos necessários.

Analisemos agora os dois circuitos referidos na pag. anterior sob o ponto de vista de tempo de execução da operação.

No 1º caso (circuito adicionador paralelo com ripple carry) quando o comprimento em bits dos números a adicionar aumenta, o tempo de propagação do carry pode tornar-se importante:

utilizando circuitos lógicos de tecnologia TTL, em que o tempo de atraso da entrada para a saída será da ordem dos 30 ns por cada Full adder, isto é, para números com 16 bits de comprimento, p. ex. um tempo de execução da operação de cerca de 480 ns.

No 2º caso (circuito somador paralelo com "look ahead carry generator") o tempo de execução da operação é independente do comprimento dos números a operar e será da ordem de grandeza de 60 a 80 ns.

Note-se no entanto que a complexidade do circuito aumenta fortemente com o comprimento em bits dos números a somar.

Como solução intermédia pode optar-se por blocos adicionadores de 4 bits com "look ahead carry generator" e ligar estes blocos entre si em cascata, ou seja, com propagação de sinal de carry.

Neste caso, e para os mesmos 16 bits de comprimento dos números a somar, o tempo necessário à operação andará pela ordem dos 100/120 ns

2.2. Operações em código BCD

2.2.1. Soma paralelo

Os somadores e subtractores de numeros em código BCD não são grandemente diferentes dos circuitos para códigos binários descritos atrás. A complexidade adicional advém do facto de os numeros a operar serem expressos e manipulados em BCD.

Vejamos como efectuariamos a soma de 2 numeros em BCD:

DECIMAL	BCD		
485	0100	1000	0101
+261	0010	0110	0001
746	0110	1110	0110
		14_{10}	6_{10}

Após as operações de soma em cada uma das posições decimais conforme feito acima, é necessário em primeiro lugar detectar a existência de carry decimal resultante da soma de quaisquer dois dígitos.

Neste caso, na posição 10^1 origina-se uma carry decimal que terá que ser somada na posição 10^2.

Em seguida, e nos dígitos em que houve carry decimal, há também que corrigir o resultado da soma, ou seja, no exemplo acima corrigir $14_{(2)}$ para $4_{(2)}$.

Para detectar a existência de carry ter-se-á que implementar um circuito que descodifique os estados inválidos (10, 11, 12, 13, 14, 15) mais os casos em que existe carry binária (16, 17, 18, 19).

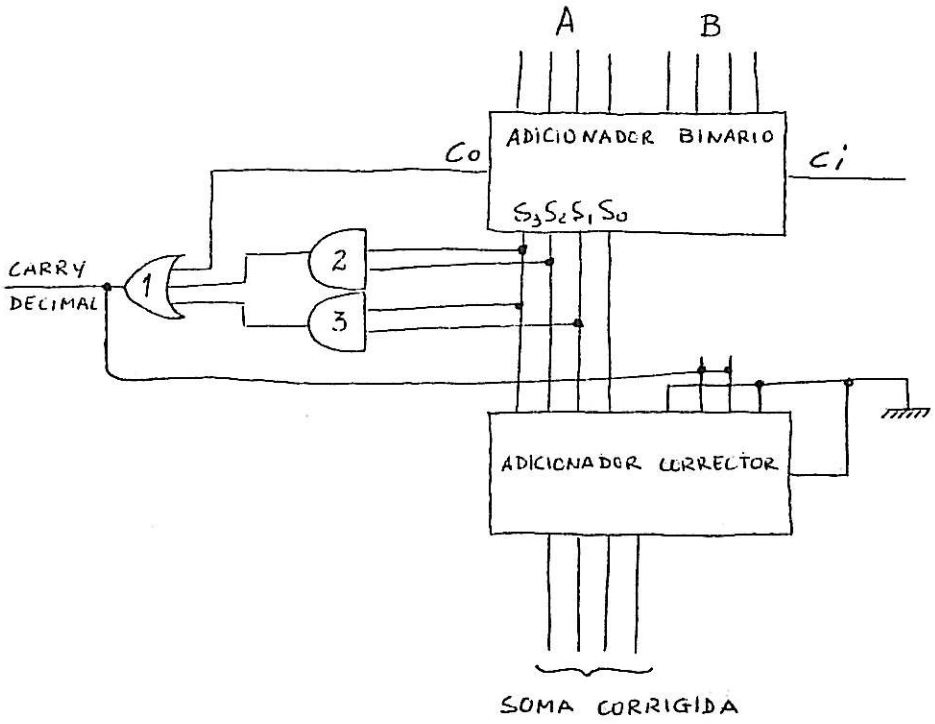
Caso alguma destas situações ocorra, somar-se-á 1 na posição decimal seguinte (mais significativa).

Para corrigir o resultado da soma nos casos acima referidos teremos também que subtrair 10 ao resultado obtido.

Achando o complemento de 2 do numero 10 (que é $6_{10} = 0110_2$) e o somarmos na posição a corrigir, ignorando

erential carry que apareça obtemos o resultado corrigido, isto é, subtraido de 10.

Vejamos a implementação que executa este algoritmo



BLOCO ADICIONADOR BCD

As gates 2 e 3 descodificam os estados invalidos 10, 11, 12, 13, 14 e 15. A gate 1 fat o "ou" logico desta descodificação com o sinal de Co do adicionador, que existiá nas somas cujo resultado seja 16, 17, 18 ou 19, e dá portanto origem á carry decimal.

Esta carry também vai actuar nas entradas do adicionador corrector, comandando a soma de 0110 ou 0000 ao resultado do primeiro adicionador, conforme seja ou não necessário corrigir.

Como é evidente, para somar numeros, codificados em BCD, com várias posições decimais, vão há mais que ligar em cascata vários blocos adicionadores como o apresentado acima.

2.2.2 Subtractor paralelo

Para efectuar subtracções em código BCD, a filosofia a seguir é similar à utilizada nas subtracções em código binário, salvo que, neste caso é conveniente utilizar complementos de 9 para representar os numeros negativos em vez de complementos de 1.

Utilizando os complementos de 1 para representar os numeros negativos, o algoritmo de execução da subtracção viria grandemente complicado, embora seja exequivel.

Trataremos portanto a subtracção em complementos de 9:

Seja a operação $A - B$ em que A e B são numeros com n digitos decimais.

$$A - B \Rightarrow A + (10^n - 1) - B = 10^n + (A - B - 1)$$

Note-se, em primeiro lugar que, dispondo nós de n digitos decimais para representar os nossos numeros, temos a possibilidade de discernir 10^n numeros distintos. Desde que pretendamos representar numeros positivos, e numeros negativos (pelo complemento a 9's do numero positivo correspondente), teremos necessáriamente que partilhar as 10^n representações distintas pelos numeros positivos e negativos.

Isto conduz-nos a que todos os numeros até $\underbrace{49\ldots9}_{n}$ sejam numeros positivos.

De $500\ldots0$ até $999\ldots9$ são numeros negativos (são exactamente os complementos a 9 dos numeros positivos de $0\ldots$ a $49\ldots9$).

Portanto e resumindo o exposto, quando representamos numeros positivos e negativos (negativos através dos complementos de 9's) apenas podemos representar numeros positivos e negativos de 0 até $\underbrace{49\ldots9}_{n \text{ digitos}}$ usando palavras com n digitos de comprimento.

Exemplo: Para representar numeros de -99 a $+99$ necessitamos de 3 digitos decimais.

No entanto esses 3 digitos decimais, permitem além disso, representar numeros de -499 até $+499$.

Continuemos então com o algoritmo da subtracção

$$A - B \Rightarrow 10^n + (A - B - 1)$$

a) Daqui deduz-se que se $A-B$ for negativo, não vai existir sinal de transporte e o resultado continuará expresso em complementos de 9.

b) Se $A-B$ for positivo então $10^n + (\underbrace{A-B}_{\text{nº positivo}} - 1)$ dá origem a um sinal de transporte decimal, que se deve somar ao resultado para obter o valor correcto.

$$10^n + (A - B - 1) = 1 \times 10^n + (A - B) - 1$$

Para se obter o valor correcto vamos: - despertar o 1×10^n

- Somar 1 ao restante

ou seja, somamos o sinal de transporte decimal de peso 10^n na posição 10^0.

Exemplo:

$A = 24 \Rightarrow 0010\ 0100$

$B = 17 \Rightarrow 0001\ 0111$

$-A \Rightarrow 75 \Rightarrow 0111\ 0101$

$-B \Rightarrow 82 \Rightarrow 1000\ 0010$

(Como se vê, utilizando dois digitos decimais não representamos numeros superiores a 49, nem inferiores a -49)

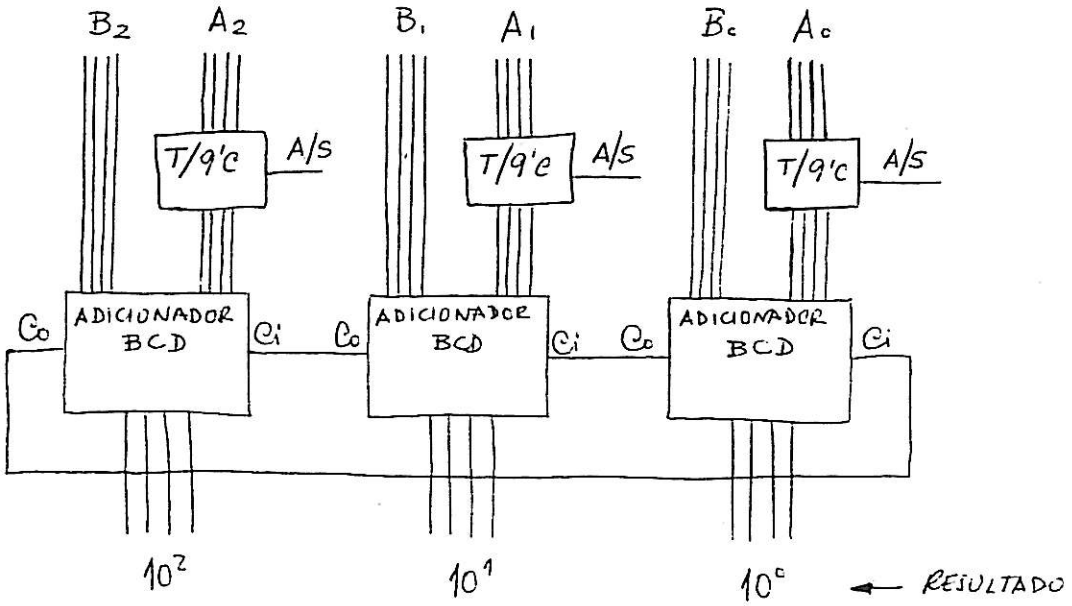
Efectuemos então A - B

24	24	0 0 1 0	0 1 0 0
-17	+82	1 0 0 0	0 0 1 0
7	106	1 0 1 0	0 1 1 0
	↳ 1	0 1 1 0 ←	CORRECÇÃO
	07	1 0 0 0 0	0 1 1 0
		↳	1
	RESULTADO CORRECTO:	0 0 0 0	0 1 1 1

Leja agora B - A

17	17		
-24	+75	0 0 0 1	0 1 1 1
-7	. 92	0 1 1 1	0 1 0 1
	⇓	1 0 0 0	1 1 0 0
	-7		1 0 1 0 ← CORRECÇÃO
	RESULTADO EM COMPLEM. DE 9's:	1 0 0 1	0 0 1 0

Um circuito que executa este algoritmo de subtracção assim como o de soma pode ser do tipo seguinte:



O bloco adicionador básico do circuito anterior, não é mais do que o adicionador BCD descrito atrás.

O Bloco T/9'C é um bloco que, mediante o controle da entrada A/S (Adiciona/subtrai), apresenta à saída a mesma palavra que a entrada ou apresenta o seu complemento a 9's.

A implementação deste circuito será imediata a partir de tabelas de verdade simplificadas.

Note-se ainda que a não existência de end-around-carry indica que o resultado é um número negativo, o que pode ser utilizado para comandar a complementação a 9's do resultado se se desejar este sob a forma de sinal e valor absoluto.

Deve salientar-se aqui uma das vantagens do código excesso de 3 na execução de operações aritméticas, que é exactamente a facilidade com que se obtém, nesse código o complemento a 9's de um número. Conforme se pode verificar, basta para isso, complementar os bits do número

Outra das vantagens das operações em código exs-3 é o facto de, quando se adicionam dois dígitos expressos nesse código, obtermos uma carry na posição 2^4 sempre que e só se essa carry também existir na soma decimal, ou seja, a carry do código exs-3 coincide com o sinal de transporte decimal. Neste caso, não seria necessário descodificar os estados inválidos como foi no somador BCD. Continua no entanto a ser necessário a correcção dos resultados quando é originada uma carry.

2.3. Somadores / Subtractores utilizando contadores de impulsos

Baseados numa filosofia de operação totalmente diversa dos circuitos estudados anteriormente, pode-se implementar um circuito somador/subtractor com contadores de impulsos que sigam o código crescente e/ou código decrescente (contadores UP e UP/DOWN).

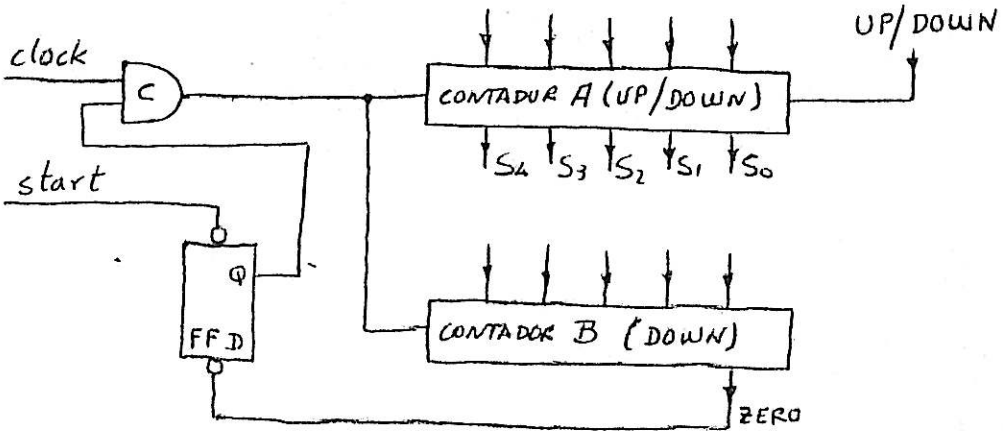
A filosofia básica está apoiado no seguinte método; seja a soma

35 + 12 a efectuar em código binário

- Coloquemos um contador binário, através das entradas paralelo no estado 35.
- Se, a seguir, fizermos actuar na entrada de "clock" 12 impulsos, o contador passará ao fim destes, para o estado 47 o que é exactamente a soma de 35 + 12.

Se o contador for "UP/DOWN" e o posicionarmos para contagem "DOWN", após os 12 impulsos ele ficará no estado 23 ou seja 35 − 12.

Vejamos um exemplo mais aplicado;



Operação: 1- Façamos transferir para os contadores A e B, os dois numeros a somar, através das entradas paralelo. O FF D encontra-se no estado Q=0, logo a gate C está bloqueada.

2- Ao ser dado o impulso de "START", o FFD passa ao estado $\varphi=1$, e vai portanto desbloquear a gate C, o que permite a passagem do sinal de "clock" para ambos os contadores.

Por cada impulso que surge, o contador A adiciona <u>um</u> ao valor do seu estado, e o contador B diminui <u>um</u> (B é um contador "DOWN").

3- Quando o contador B atingir o estado <u>zero</u> o FFD é "resetado" (i.e. passa a $\varphi=0$) e a gate C fica de novo bloqueada.

4- O Resultado da operação fica presente nas saídas do contador A.

Para efectuar subtracções com este circuito, não haveria mais do que comandar o contador A para contagem "DOWN", ou seja, por cada impulso recebido, este diminuiria <u>um</u> ao valor do seu estado.

Note-se que se tentarmos efectuar uma subtracção, com este circuito, em que o subtractor seja maior que o subtraendo (nºs positivos), o resultado aparece expresso em complementos de 2's.

Isto aconselha a que, com este tipo de circuitos, representemos os nºs negativos (caso seja caso disso) pelo seu complemento de 2's.

Note-se também que o contador B (tipo "down"), poderia ser substituido por um contador "UP", desde que tivessemos o cuidado para que o nº lá armazenado e que vai ser <u>somado ou subtraido</u> ao numero do contador A, seja substituido pelo seu complemento de 2's. Nesse caso, o nº de impulsos necessários para o contador B (com <u>n</u> bits de comprimento) passar de $\underbrace{(2^n-B)}_{\text{2's COMPLEM. DE B}}$ para $\underbrace{2^n}_{\underbrace{10\ldots0}_{n\text{ bits}}}$, seria $2^n-(2^n-B)=B$, que é o desejado.

Para efectuar operações em outros códigos, diferentes do binário, segundo este método, apenas haveria que substituir os contadores binários por contadores que seguissem esses códigos.

O esquema e modo de funcionamento atrás descrito, apresenta apenas a filosofia geral e básica deste tipo de operadores.
Repare-se portanto, que como tal, apresenta certo tipo de inconvenientes, nomeadamente no que respeita ao tempo necessário à execução da operação.
Como este tempo vai ser o necessário ao aparecimento de tantos impulsos de "clock" quanto o valor do nº a somar ou subtrair, e a frequencia do sinal de "clock" para tecnologia TTL (por exemplo), está limitada superiormente a cerca de 10MHz, caso pretendamos somar numeros grandes, este tempo pode ser incomportável.

Exemplo: Seja o comprimento max. em bits dos operandos, 8.

Seja então a operação 180 + 255
Como para $f = 10MHz \Rightarrow$ periodo do sinal de clock:

$$T = 100\,ns$$

a operação demoraria a executar:

$$255 \times 100\,ns \approx 26\,\mu s$$

tempo este que tornaria a máquina extremamente lenta.

Para obviar a este inconveniente, podemos optar por um esquema um pouco mais elaborado, mas que obedece à mesma filosofia de base.

- Admitamos, tal como no exemplo acima, que pretendemos operar nºs com um comprimento em bits de 8. (logo de 0→256)
- Subdividamos os contadores A e B do esquema anterior em subcontadores A_1, A_2, B_1 e B_2, cada um deles com 4 bits.
- Em seguida, para executar a operação, iremos actuar a entrada de clock de um modo semelhante ao circuito anterior

- Quando o contador A1 atinge o estado 15 (1111), abrimos a gate 1. Para este efeito, podemos descodificar este estado ou então aproveitar uma das saídas dos contadores integrados que vai a "HIGH" no estado 15, exactamente tal como desejamos.
- Caso este contador receba novo impulso do sinal de "clock", (que portanto o vai fazer transitar do estado 1111 para 0000), este impulso é também injectado na entrada do contador A2, que portanto soma 1 ao valor que lá existia.

2- Quando o contador B1 atinge o estado 0000 (e isso é indicado pela ida a "HIGH" da saída "ZERO", fazemos o "reset" do FF1; o que vai bloquear o envio de impulsos de clock para estes contadores (A1 e B1), mas, e aqui reside a diferença entre este circuito e o anterior, faz simultâneamente o "set" (Q=1) do FF2, que por sua vez agulha os impulsos do sinal do "clock" agora para os contadores A2 e B2.

Idêntico procedimento se vai agora desenrolar nestes dois contadores.

3- Após o contador B2 atingir -0000- e fazer o "reset" do FF2, a operação está concluida.

Para efectuar operações de subtracção com um circuito deste tipo, teriamos que provar, que os contadores A1 e A2 fossem "UP/DOWN", assim como um circuito combinatório similar às gates 1 e 2 deste circuito, que originaria um impulso de "CARRY" da subtracção, neste caso agulhado para o contador B2 (quando o contador A1 passasse do estado 0000 para 1111).

A grande vantagem deste circuito, em relação ao anterior, é a sua maior rapidez na execução das operações,

Enquanto que num circuito similar ao inicial, para somar ou subtrair numeros com 8 bits de comprimento, haveria que prever um tempo de execução da operação da ordem de 256 × periodo do "clock" ≈ 256 × 100 ns ≈ 26 μs, tal como já vimos, com um circuito do tipo agora apresentado esse tempo seria de

$$16 \times 100\,ns + 16 \times 100\,ns = 3.2\,\mu s$$

isto é, reduzido cerca de 10 vezes. Para numeros maiores, a diferença ainda é mais notória.

2.4. Multiplicação em código binário

Similarmente á tabuada da multiplicação em decimal, podemos constituir, também no código binário uma tabuada de multiplicação:

$$0 \times 0 = 0$$
$$0 \times 1 = 0$$
$$1 \times 0 = 0$$
$$1 \times 1 = 1$$

Vejamos ainda duas caracteristicas fundamentais desta operação:

1- $4 \times 6 \equiv 4 + 4 + 4 + 4 + 4 + 4$

ou seja, a multiplicação é uma sequência de adições.

2- Deslocando um nº binário de uma posição para a esquerda, equivale a multiplicá-lo por 2, no que é similar ao que se passa no código decimal.

$0\ 1\ 3$ $\qquad$ $0\ 1\ 1\ 0 \equiv 6_{10}$

$1\ 3\ 0 = 13 \times 10$ $\qquad$ $1\ 1\ 0\ 0 \equiv 6 \times 2 = 12_{10}$

Estas duas propriedades vão ser de grande utilidade na definição do algoritmo da multiplicação a executar em circuitos aritméticos.

Vejamos em 1º lugar, como é que se executa a operação de multiplicação manualmente.

```
  14          1110
 x23         x1010
 ---         -----
  42          0000
 28          1110
 ---      0000
 322     1110
         --------
         10001100
```

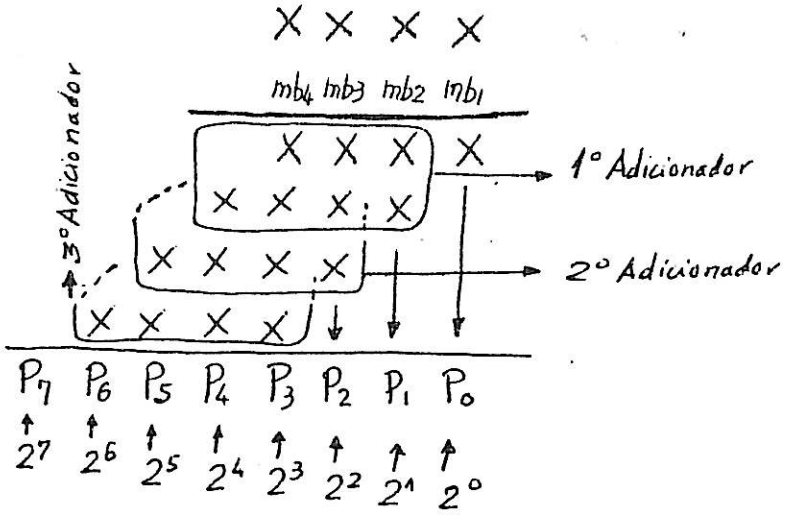
Como se pode ver:

1- Na multiplicação binária, cada um dos produtos parciais, não é mais do que a multiplicação por 0 ou 1 do multiplicando, i.e., é o próprio multiplicando ou zero.

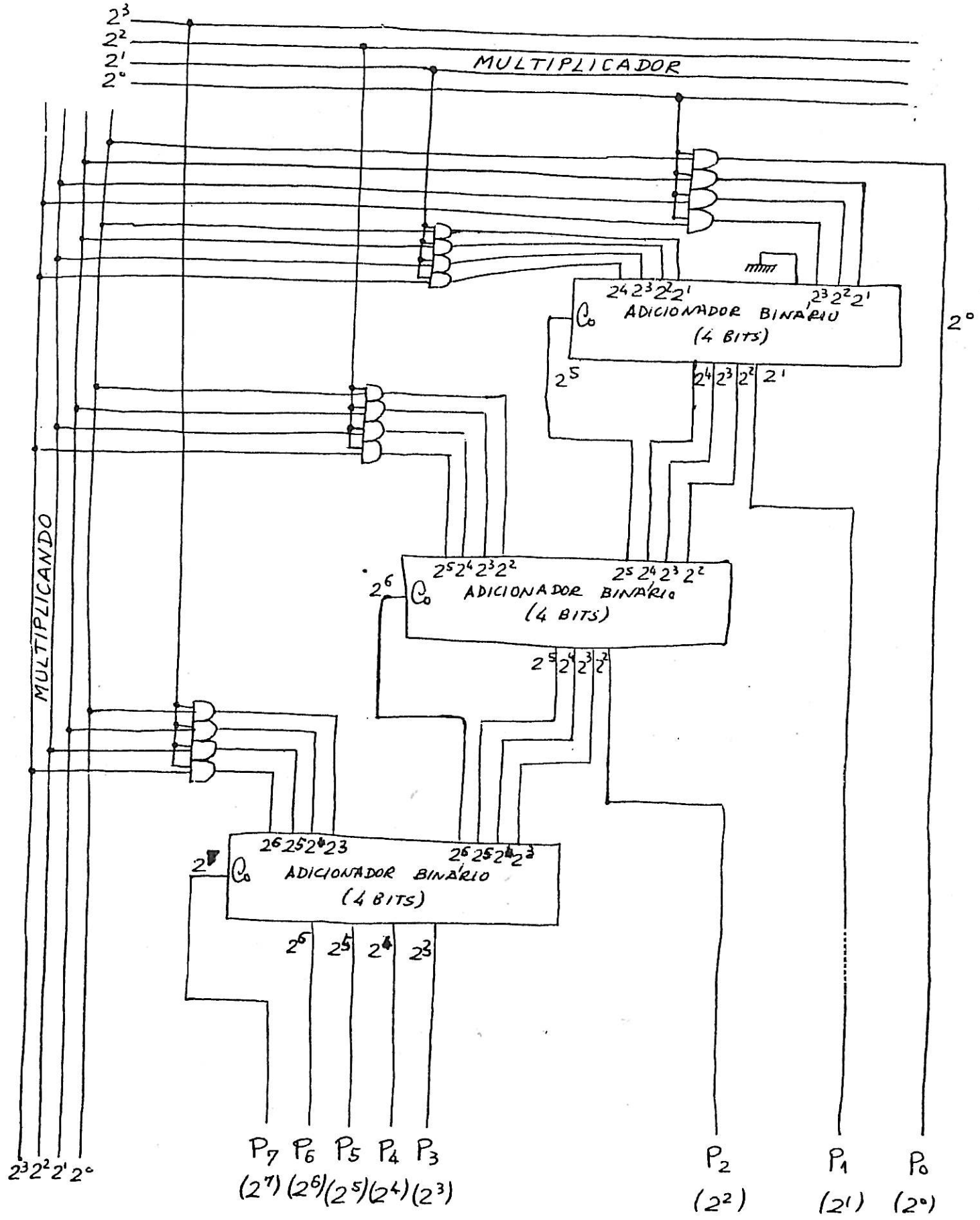
2- O resultado final obtem-se somando todos estes produtos parciais, sendo cada um deles deslocado de uma posição para a esquerda em relação ao anterior.

Vejamos agora como implementaríamos circuitos aritméticos que permitam executar esta sequência de operações elementares.

2.4.1. Multiplicador paralelo



O algoritmo a executar é o apresentado à esquerda. Vamos necessitar de 3 adicionadores de 4 bits, de modo que aguilhando convenientemente os dados, obtenhamos a execução das operações pretendidas.



Conforme se pode ver do algoritmo da pag. anterior, Po obtem-se directamente do 1º bit do multiplicador (ou é zero ou é o valor do 1º bit do mesmo conforme o 1º bit do multiplicador é zero ou um)

Seguidamente, vamos somar os restantes bits do multiplicando vezes o 1º bit do multiplicador (mb1), com o produto do mesmo multiplicando pelo 2º bit do multiplicador (mb2), sendo esta parcela deslocada uma posição à esquerda.

Procedemos assim sucessivamente, até obter o resultado final.

Chama-se a atenção para os deslocamentos relativos dos diversos produtos parciais em relação à soma corrente com que se vão adicionar.

Nota-se também que, como resultado de cada uma das somas parciais pode originar-se uma "carry", que obviamente deve ser considerada na soma seguinte.

Tal como nos circuitos somadores paralelo, a vantagem evidente desta configuração é ser bastante rápido, sendo no entanto bastante complexo (logo caro) quando o nº dos bits dos numeros a operar aumenta.

Nota-se ainda que o tempo de execução da operação, não é totalmente independente do comprimento em bits dos numeros a operar, pois há que executar sequencialmente um nº de operações parciais, que aumenta com o comprimento dos operandos em bits, e o resultado só é válido após todas estas se terem executado.

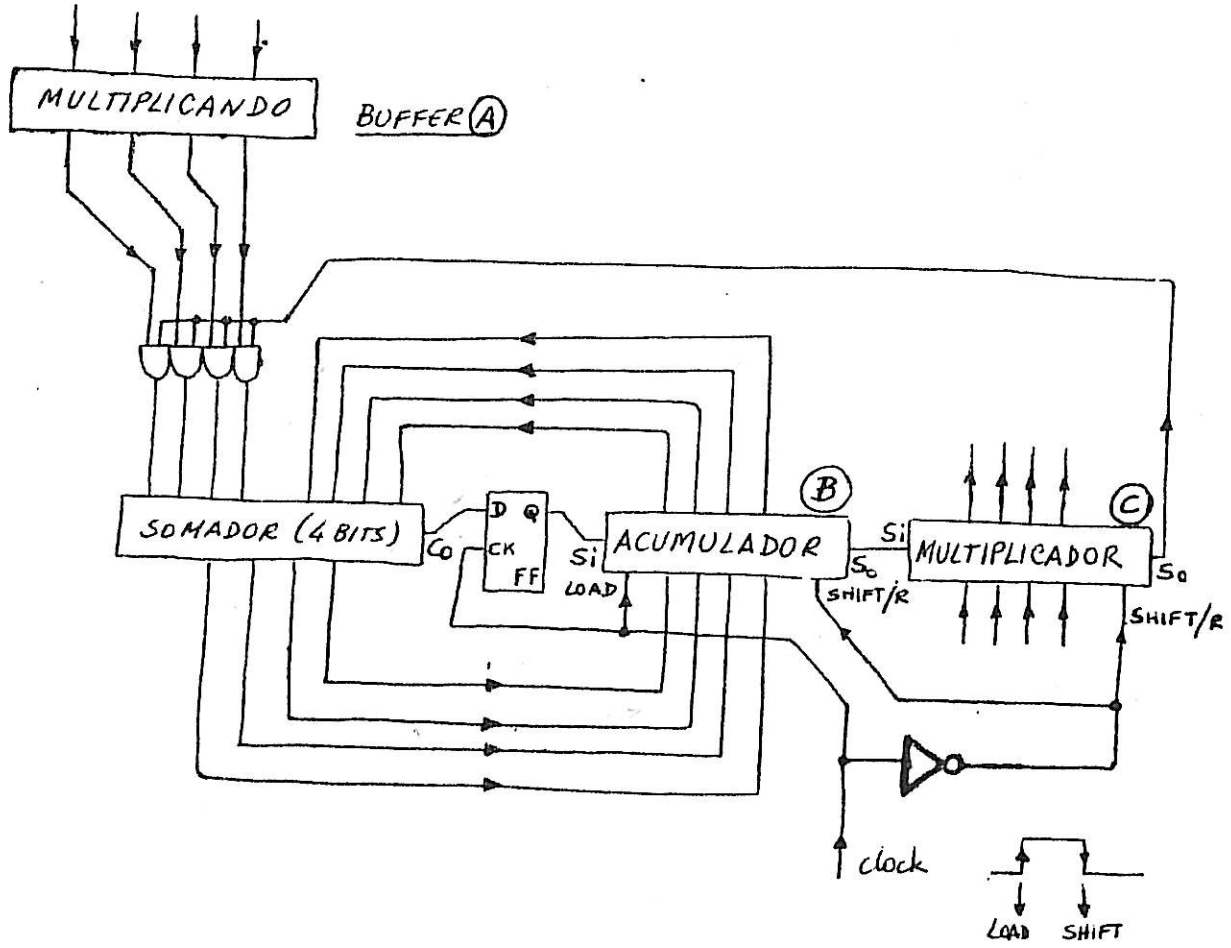
Para circuitos adicionadores de 4 bits, usando tecnologia TTL e configurações tipo "look ahead carry", e para uma unidade multiplicadora de numeros com 4 bits de comprimento, o tempo de execução da operação será da ordem de 130 ns

(10 ns de atraso nas gates por onde passa a primeira parcela + 3×40 ns de tempo de execução dos adicionadores.

2.4.2. Multiplicador série

Como base deste tipo de circuito, encontra-se o algoritmo de execução da multiplicação já descrito, notando-se apenas que, agora, as diversas operações elementares vão efectuar-se de um modo sequencial e sincronizadas por um sinal de "clock".

Isto conduz a que o problema fique reduzido a encaminhar o multiplicando retes cada um dos bits do multiplicador, para um adicionador paralelo de n bits, (n = comprimento max. em bits do multiplicando), onde se vai somar com o valor existente num acumulador, que vai armazenando as somas correntes dos produtos parciais:



Repare-se em 1º lugar que enquanto no algoritmo usual, cada produto parcial era deslocado uma posição para a esquerda antes de ser somado, neste circuito deslocamos a soma parcial de uma posição à direita antes da mesma soma, o que é perfeitamente equivalente.

Antes de observarmos como evolui o sistema, note-se o seguinte:

- Como era de esperar, o produto final terá um comprimento duplo dos operandos. Isso poderia em 1ª aproximação conduzir a que o acumulador do resultado devesse ter duplo comprimento.

No entanto, observemos que, após a obtenção de cada um dos produtos parciais, um bit do multiplicador deixa de ser necessário (pois já foi operado e não volta a ser-lo). Logo, após cada um dos produtos parciais ser obtido e somado, podemos libertar uma posição do acumulador do multiplicador.

Em contrapartida, se observarmos os produtos parciais verificamos que o 1º tem o comprimento do multiplicando. A soma do 1º com o 2º pode ter mais um bit (eventual "carry"). A soma desta "soma corrente" com o 3º produto parcial, pode ter mais um bit que a anterior e assim sucessivamente.

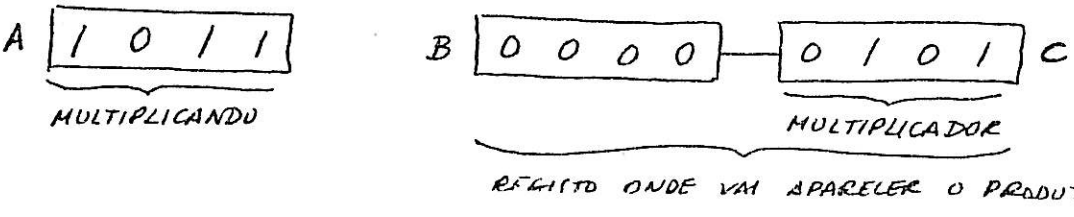
Isto conduz-nos à ideia de que podemos partilhar o acumulador do multiplicador entre este e os resultados parciais e final.

É exactamente isto que fazemos ao ligar a saída série do acumulador, à entrada série do registador de deslocamento do multiplicador, pois os resultados parciais vão-se expandindo para o registador C à medida que este vai ficando vazio, como consequência do deslocamento do multiplicador para a direita.

Vejamos então agora como evolui o sistema ao executar uma operação:

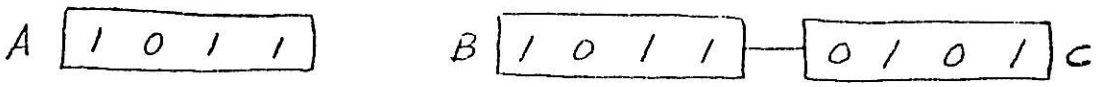
1- Armazenamos nos registos A e C os números a operar. Em B (Acumulador das somas correntes dos produtos parciais) armazenamos zero.

Sendo por exemplo A = 11 e C = 5 ficará

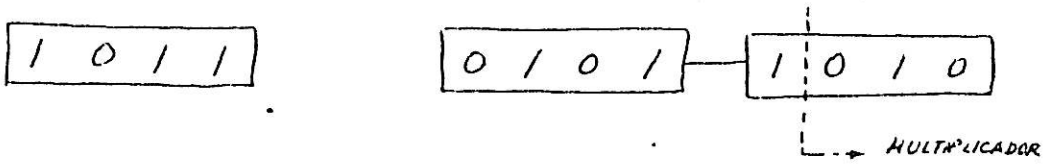


2- Dado que o bit menos significativo, ou seja, mais à direita do multiplicador é 1, o multiplicando é adicionado ao conteúdo de B (que é zero neste caso).

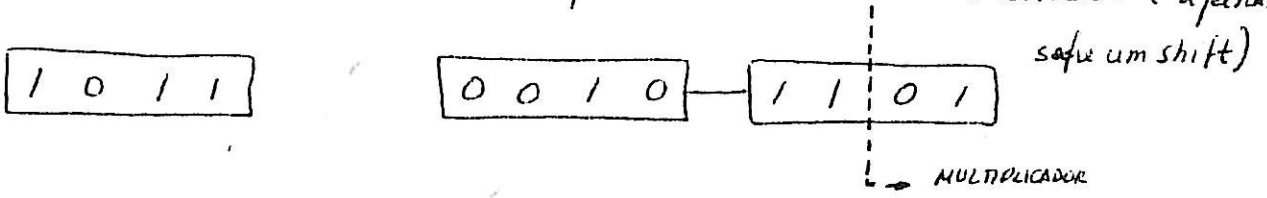
No flanco de subida do sinal de "clock", o resultado desta soma é transferido para B, substituindo portanto o valor que lá se encontrava



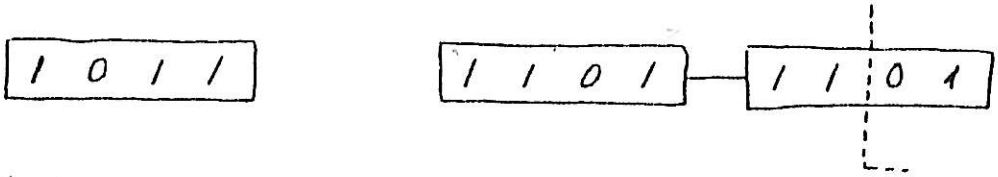
3- Seguidamente, no flanco de descida do sinal de "clock", os registos C e B são deslocados de uma posição para a direita.



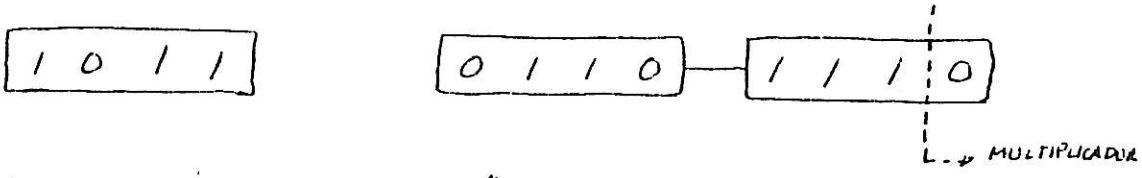
4- Como agora, o bit que nos é apresentado pelo multiplicador é 0, o somador vai adicionar 0000 ao conteúdo de B, armazena este valor no acumulador B e depois desloca uma posição para a direita o conteúdo de B e C: (ou seja o valor de B não é alterado e apenas sofre um shift)



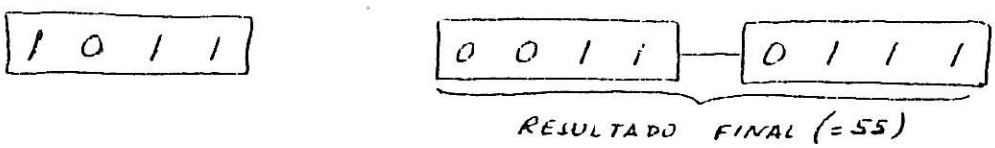
5- Após a próxima subida do "clock" efectua-se o armazenamento em B do resultado de mais uma soma;



6- Após a descida do "clock" efectua-se novo deslocamento:



7- Nova subida e descida do "clock":



Como foram injectados 4 períodos do sinal de "clock", a operação está concluida. O Resultado final encontra-se acessível nos acumuladores B e C, tendo o multiplicador sido destruido.

Note-se a existencia de um FF tipo D, que armazena uma eventual "carry" na posição 24 da soma, sincronamente com o flanco de subida do "clock", o que permite que, aquando do deslocamento para a direita do acumulador, esta informação fique armazenada na posição mais significativa do mesmo, sendo portanto considerada na próxima operação de soma.

Tempo de execução da operação

As considerações sobre o tempo de execução da operação vão aqui depender de várias parâmetros.

Vejamos em 1º lugar as limitações do sinal do "clock":

i- Após a subida do sinal de "clock", vamos executar a transferencia paralelo do resultado da soma para o acumulador B. Logo, antes de ser dada ordem para deslocar para a direita, temos que garantir que a transferência está executada.

ii- Após a descida do sinal de "clock", vamos executar uma operação de deslocamento, um produto parcial e uma nova soma.

O resultado desta só pode ser armazenado em B após a execução destas operações.

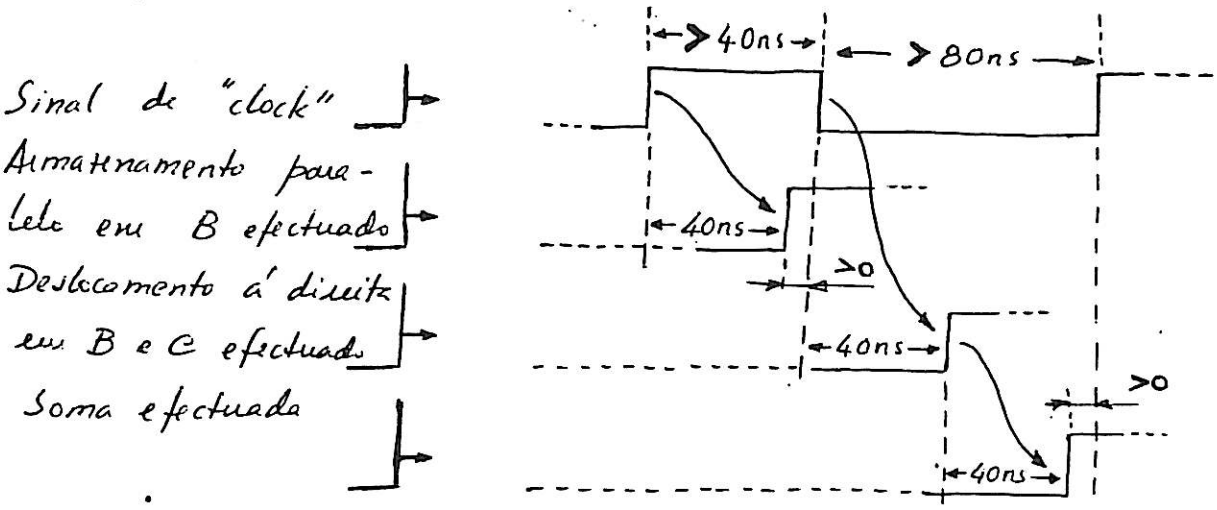
Portanto o sinal de "clock" vai depender não só da tecnologia utilitada, mas também das configuraçoes dos shift-registers, e do adicionador.

Exemplifiquemos, para o circuito atrás descrito (operandos com 4 bits de comprimento) e considerando que:

- o bloco adicionador de 4 bits é do tipo "look ahead carry", logo tempo de execução da soma, após estabilitação dos numeros à entrada: ~ 40 ns

- os registos B e C, são do tipo entrada paralelo, saída paralelo / saída série com deslocamento para a direita, em que o tempo necessário ao armazenamento paralelo é da ordem dos 40 ns, assim como o tempo necessário à execução do deslocamento da informação.

Então:



Como se depreende do diagrama temporal apresentado, o período do sinal de "clock" deve ser superior a 120 ns, (como segurança não devemos permitir que seja inferior p. ex. a 160 ns) e com a relação HIGH/LOW (*) semelhante à apresentada acima.

Deste modo, a execução do produto de dois números num circuito semelhante ao apresentado, demoraria

$$4 \times 160\,ns \simeq 640\,ns$$

Note-se que para operar números com um comprimento em bits maior, (p. ex. 16 ou 32) além de serem necessários mais períodos do sinal de "clock" para completar a operação, o bloco somador pode ser mais lento e nesse caso o período do "clock" teria também que ser ligeiramente maior.

(*) usualmente chamada "duty cycle".

2.5. Divisão em binário

Uma vez que a divisão é a operação inversa da multiplicação, ela deve poder ser executada através de sucessivas subtracções, sendo o nº de vezes que esta pode ser feita, igual ao quociente.

Exemplo:

40 : 9	4 0	Quociente
	− 9	
	3 1	1
	− 9	
	2 2	1+1
	− 9	
	1 3	1+1+1
	− 9	
	4	1+1+1+1
	Resto	Quociente

No entanto, se atendermos à propriedade de que um número deslocado uma posição à esquerda vem multiplicado pela base de numeração, e à direita vem dividido, podemos executar divisões através de um algoritmo mais rápido:

Seja → 1 4 9 0 : 7

	A adicionar ao Quociente	QUOCIENTE
1 4 9 0		0 0 0
− 7 0 0		
7 9 0	100 ⇒	1 0 0
− 7 0 0		
9 0	100 ⇒	2 0 0
− 7 0		
2 0	10 ⇒	2 1 0
− 7		
1 3	1 ⇒	2 1 1
− 7		
6	1 ⇒	2 1 2

Em vez de subtrairmos o nº 7, 212 vezes de 1490 o nosso procedimento foi o seguinte:

- Deslocámos o divisor (neste caso 7) duas casas à esquerda, (ou seja (x) por 100) e assim efectuámos subtracções até ser possível, isto é até o resultado da mesma ainda ser positivo. Neste caso duas vezes.
Como o subtractor foi multiplicado por 100, por cada subtracção efectuada, o resto é incrementado de 100.
Após esta situação deslocamos o subtractor uma posição para a direita, (ou seja, fica apenas (x) por 10) e repetimos as operações. Então, por cada subtracção possível efectuada, incrementamos o quociente 10 unidades.

Código binário

Vejamos agora quais as consequências para este algoritmo, quando pretendermos trabalhar em código binário e efectuar o cálculo de um modo automático.

1- Dado operarmos em código binário, cada um dos bits do quociente só pode ser 0 ou 1, ou seja, após cada subtracção efectuada, vão só possível efectuar outra, sem deslocar o divisor uma posição para a direita.
Logo a sequência a efectuar será apenas:
Tentar subtrair e seguidamente deslocar o divisor à direita.
Se a operação pôde ser feita, colocamos 1 na posição correspondente do quociente. Caso contrário, colocamos 0.

2- No exemplo manual anterior, escolhemos 700 (ou seja 7x100) para ser o 1º subtractor.
No entanto, dado que a máquina não tem a possibilidade de efectuar essa escolha senão por um método exaustivo e sistemático, deveríamos pensar que se iniciassem as tentativas de subtracção com o divisor deslocado para a esquerda o mais possível, e depois ir tentando subtrair e deslocando para a direita, até conseguir efectuar a 1ª subtracção. Isto

portanto colocando zeros nas posições mais à esquerda do quociente, até esta posição em que colocaria o primeiro 1.

Os inconvenientes deste procedimento, são que haverá que executar mais passos de operação, desnecessários, uma vez que apenas nos conduzem a zeros não significativos no quociente, assim como iríamos ocupar posições do registo do quociente com estes zeros sem significado, perdendo portanto possibilidade de aproximação do resultado (caso o comprimento max. do quociente seja fixo, como é natural).

Para obviar a este inconveniente é costume exigir que, os números a dividir estejam alinhados, sejam da mesma ordem de grandeza (isto é, não difiram mais do que 2°) e ainda que o divisor seja maior que o dividendo.

Esta última condição vai, evidentemente, conduzir a que o quociente tenha apenas parte fraccionária, i. é. seja < 1.

Este pequeno pré-tratamento dos operandos permite, no entanto, começar a operação imediatamente (sem necessidade de determinar automáticamente até onde se deve deslocar o divisor para começar) e além disso, a primeira tentativa de subtracção, que vai ser infrutífera, serve exactamente para determinar se as condições acima enunciadas foram cumpridas.

Estas condições conduzem também a que o registo do quociente seja aproveitado de um modo mais eficaz, uma vez que as suas posições vão todas ser ocupadas com bits significativos.

3- Outra das decisões que nós tomámos ao efectuar a operação de um modo manual, foi que p. ex. após a segunda subtracção de 700 não iríamos tentar outra (que já não seria possível) e então deslocámos o divisor uma casa para a direita.

De um modo automático esta decisão só pode ser tomada "à posteriori" i. é, o circuito executaria a subtracção e só após isto, é que verificava que o resultado era negativo. Então, neste caso, há que prever que o subtractor seja de novo adicionado ao resultado (para restabelecer ou restaurar o valor inicial), e depois proce-

dem aos passos correspondentes à não possibilidade de subtracção (colocar 0 na posição correspondente do quociente e deslocar o divisor à direita).

É por causa deste tipo de operação que o método que estamos a descrever se chama usualmente "restoring method" ou seja método de restauração (do dividendo).

Métodos que conduzem a circuitos mais complicados, poderiam por exemplo, comparar o subtraendo e o subtractor e caso este fosse maior, dar sinal de operação impossível.

Vejamos então um exemplo de divisão em binário:

$160 : 28 \qquad 10100000 : 11100$

$\Downarrow$

$\underbrace{1010}_{A} \times 2^4 : \underbrace{1110}_{B} \times 2^1$

Em primeiro lugar, preparámos os operandos para ficarem conforme com as condições enunciadas na pag. anterior. Logo, o resultado obtido deverá ser multiplicado por $2^4 : 2^1 = 2^3$.
Vejamos como operaríamos manualmente:

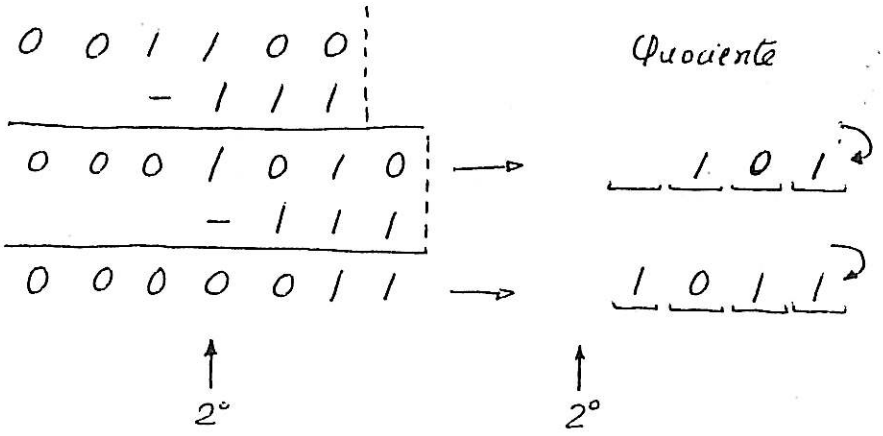
$$\begin{array}{r} 1\,0\,1\,0 \;\big\lceil 2^0 \\ -\,1\,1\,1\,0 \\ \hline \end{array}$$

Resultado $\ominus$. Logo as condições exigidas devem verificar-se. Continuemos:

$$\begin{array}{r|l} 1\,0\,1\,0 & 0\ldots \quad \text{Quociente} \\ -\,1\,1\,1 & \\ \hline 0\,0\,1\,1\,0 & \ldots \quad \_\_\_\_1) \\ -\,1\,1\,1 & \\ \hline \end{array}$$

Resultado $\ominus$. Logo 0 na posição correspondente do quociente:

$\_\_\_10)$



Note-se que o quociente deve ser (x) por 2^3 para obtermos o resultado correcto e que o resto deve ser (x) por 2^4 (pois foi por 2^4 que dividimos o dividendo antes de efectuar a operação).

Então:

Resto: $0\,1\,1\,0 \equiv 6_{10}$

Quociente: $0\,1\,0\,1_{,}1 \equiv 5.5_{10}$

Resumindo, os passos a executar, são:

1- Subtraimos o divisor do dividendo
   1a- Se o resultado é ⊕, colocamos 1 na posição mais à direita do registo do quociente.
   1b- Se o resultado é ⊖, colocamos 0 na mesma posição.

2- Deslocamos o quociente uma posição à esquerda e o divisor uma à direita (ou então deslocamos o quociente e o dividendo uma posição à esquerda o que é equivalente)

3- Repetimos os passos 1 e 2, até que:
   a- a subtracção conduza ao resultado 0 0 ... 0,
   b- a precisão requerida na operação seja atingida,
   c- todas as posições do registo do quociente estejam ocupadas.

Vejamos agora, como é que os dados se movimentam nos registos respectivos, aproximando mais à realidade do calculo automático, a operação atrás efectuada.

Notemos em primeiro lugar, que o dividendo pode ocupar duas palavras de comprimento, pois conforme vamos deslocando o divisor à direita (ou o dividendo à esquerda) estamos a fazer entrar <u>zeros</u> nas posições menos significativas do mesmo. Se, em vez destes <u>zeros</u>, considerarmos bits significativos de uma parte fraccionária do dividendo, estamos a aumentar a precisão da operação.

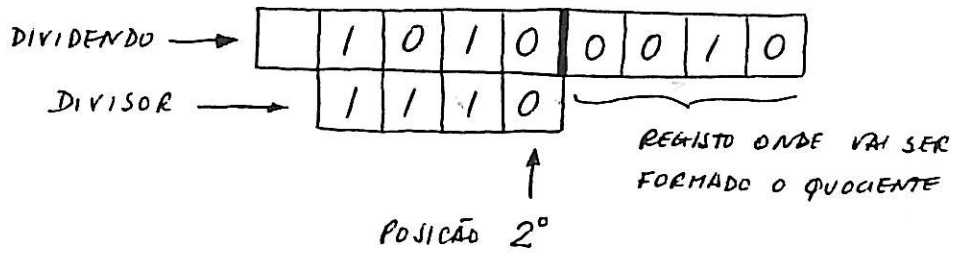
Note-se além disso que, como o quociente inicialmente não existe, e vai precisando apenas de uma posição do seu registo, por cada deslocamento do dividendo, é perfeitamente possível, tal como o foi na multiplicação, partilhar o registo do quociente entre este e uma parte fraccionária do dividendo.

Seja então 162 : 28 i.e.: 10100010 : 11100

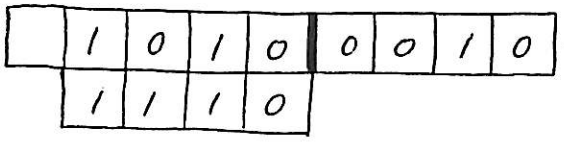
$$\underbrace{1010.0010}_{A} \times 2^4 : \underbrace{1110}_{B} \times 2^1$$

O Quociente deverá portanto ser multiplicado por 2^3 e o resto por 2^4.

1- Posição inicial dos registos:

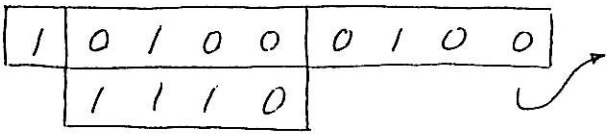


2- Iniciamos então a sequência de passos resumido na pag. anterior:



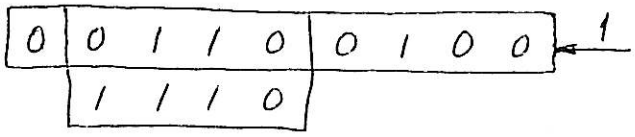
Conduz a resultado 0. Logo restauramos o dividendo e a verificação das condições está feita.

a - SHIFT LEFT

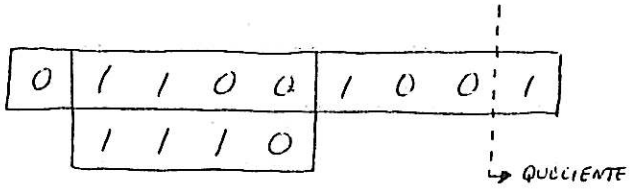


Este bit ainda não fat parti do quociente. Corresponde ao 0_9...
Se não fosse 0 as condições exigidas aos operandos não se tinham verificado e não podíamos continuar a operação.

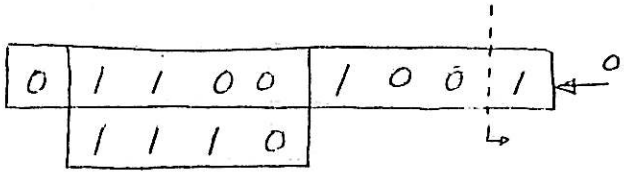
b - SUBTAI



c - SHIFT LEFT

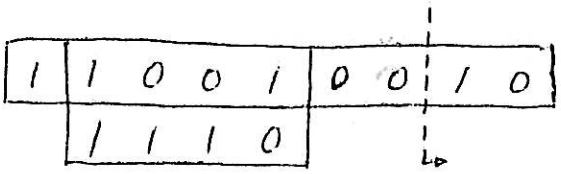


d - SUBTRAI

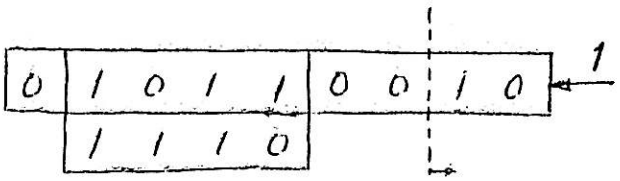


operação impossível, logo restaurou o subtraendo e colocou 0 à entrada

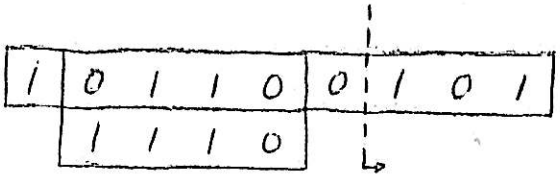
e - SHIFT LEFT



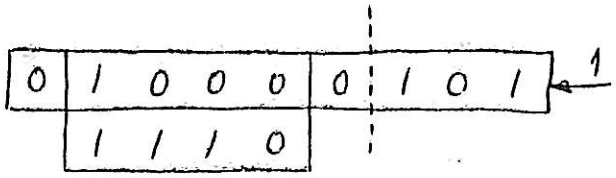
f - SUBTRAI



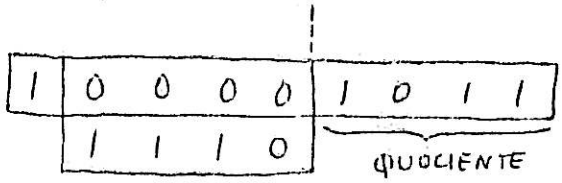
g - SHIFT LEFT



h - SUBTRAI



i - SHIFT LEFT



Note se que o resto foi deslocado à esquerda 5 vezes e como apenas devia ser (x) por 2^4, o resto real é $1000 \equiv 8_{10}$.
O quociente é $0.1011 \times 2^3 \equiv 101.1 \equiv 5.5_{10}$

Nota. Poderia-se ia, através deste circuito, continuar a operação e obter o quociente em dupla precisão.

Para isso, apenas haveria que transferir estes 8 bits do quociente para um buffer externo, ou para a memória, colocar <u>zeros</u> no registo do quociente e continuar a dividir o resto, até obter mais 8 bits significativos do quociente.

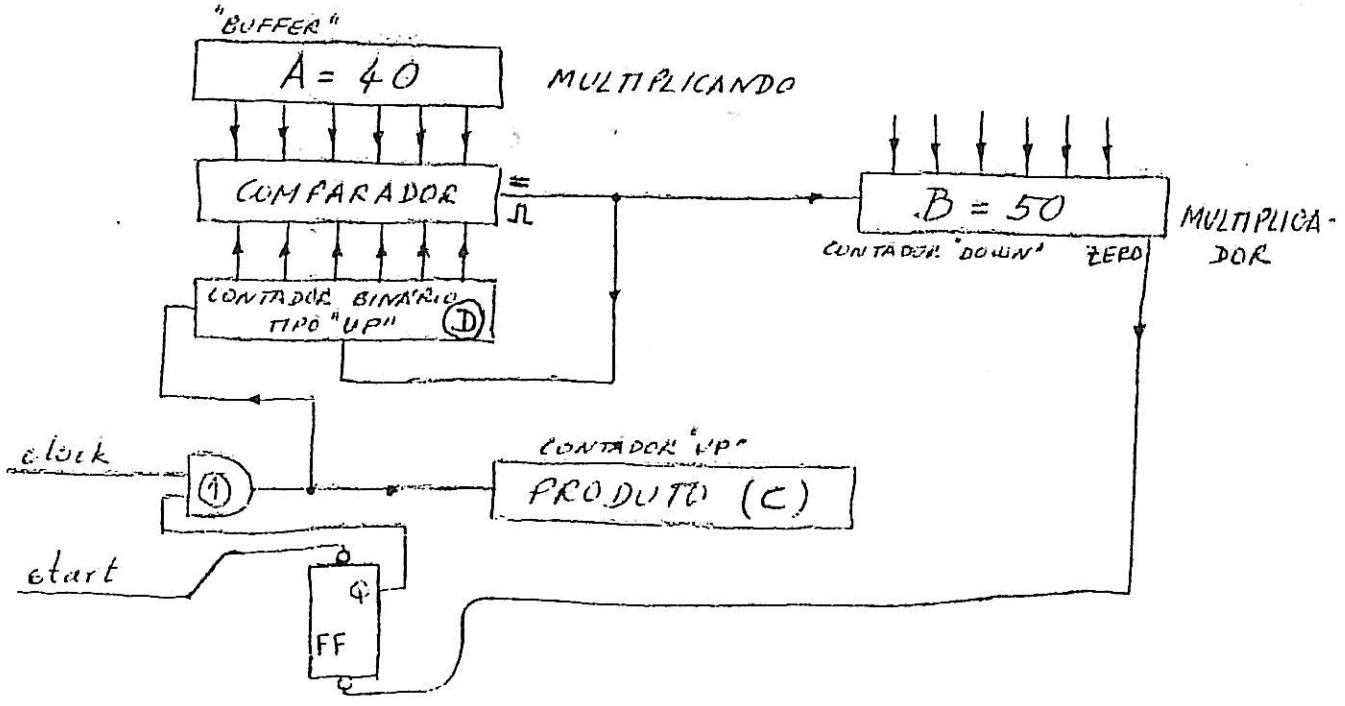
2.6. Multiplicação e divisão utilizando contadores

2.6.1 Multiplicação binária

Seguindo uma filosofia semelhante à utilizada para somadores/subtractores com contadores, podemos tentar implementar um circuito que execute o algoritmo da multiplicação.

O nosso objectivo é realizar a operação $A \times B$. Para isso, poderemos somar $A+A+A+\ldots+A$ num total de <u>B</u> vezes.

Isto conduz-nos a que devemos gerar <u>B</u> grupos de <u>A</u> impulsos e contá-los num bloco contador binário.



Operação. 1- Façamos transferir para o "buffer" A, o valor do multiplicando e para o contador "down" B, o valor do multiplicador.

2- Ao ser dado o impulso de "START", a saída Q do FF vai a HIGH, abre a gate ① e deixa o sinal de "clock" passar para o contador D e contador C (onde vai ser formado o produto dos dois números).

3- Após terem contado nos contadores C e D um nº de impulsos igual ao multiplicando, neste caso 40, o contador D está no estado 40, logo o comparador indica na sua saída = que isto aconteceu. Este impulso vai entrar no contador "down" do multiplicador e vai portanto diminuir <u>um</u> ao valor que lá se encontra (neste caso, 50). Indica portanto que só falta executar (50-1) somas.

Simultâneamente, este sinal coloca o contador D, de novo no estado <u>zero</u>. Note-se que o contador C já se encontra no estado 40, ou seja já acumulou o valor da 1ª soma.

4- Como o contador D voltou a <u>zero</u>, o processo recomeça sucessivamente durante mais 49 vezes, acumulando-se no contador C o nº total de impulsos que vão aparecendo.

5- Quando o contador B atinge zero, indicando portanto que já foram feitas as 50 somas parciais, fazemos o "reset" do FF, o que dá por terminada a operação, pois fecha a gate de passagem do sinal de "clock".

— Limitações deste circuito

Dado que, para completar a operação, é necessário fazer entrar no contador C tantos impulsos como o valor do resultado, esta configuração torna-se impraticável, sob o ponto de vista de tempo de execução, se os números a operar forem grandes.

Por exemplo, para numeros com 6 bits de comprimento, logo resultado com 12 bits (logo da ordem de 4.000 max), a operação poderia demorar, utilizando tecnologia TTL normal ($f_{max} \approx 10\,MHz$):

$$4.000 \times 100\,ns = 400\,\mu s$$

Por este motivo, implementações realistas deste tipo de circuitos são efectuadas com certo tipo de sofisticações semelhantes às apresentadas para os adicionadores com contadores.

Vejamos de um modo rápido qual a filosofia a seguir, não entrando no entanto em detalhes de circuito.

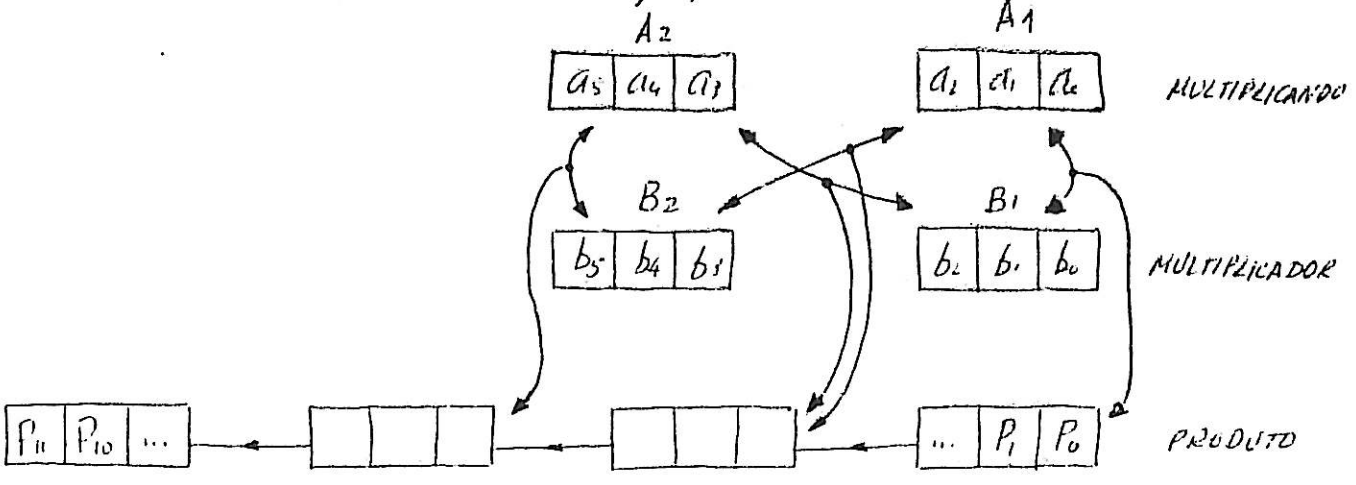
Para isso atendemos ligeiramente o algoritmo seguido: Seja em decimal, uma multiplicação, que podemos dispor no papel, do seguinte modo:

$$\begin{array}{r} 24 \\ \times\ 35 \\ \hline \end{array}$$

		5×2	5×4
	3×2	3×4	
	P_2	P_1	P_0
	10^2	10^1	10^0

Filosofia identica a esta, pode ser seguida em binário, aproveitando o circuito descrito anteriormente para executar as operações elementares. (5×4, 5×2, 3×4, ...), agulhando convenientemente os impulsos, de modo a entrarem no contador do resultado nas posições com o peso conveniente.

De uma maneira simplificada teria:



Operação. Os impulsos resultantes do produto A_1 com B_1 são acumulados nas posições menos significativas do contador produto.

Os impulsos resultantes dos produto A_1 com B_2 e A_2 com B_1 são agulhados directamente para a posição 2^3 do contador.

Os impulsos resultantes do produto A_2 com B_2 são agulhados para as posições 2^6 do mesmo contador.

Assim, a operação produto de 2 numeros com 6 bits de comprimento, que num circuito como o descrito inicialmente, demoraria, como vimos, $\approx 400\mu s$, poderia com este aperfeiçoamento, passar a demorar apenas, o tempo necessário à execução de 4 produtos, mas agora, cada um deles, com um resultado máximo de $7 \times 7 = 49$.

Logo cada uma dessas operações elementares:

$$49 \times 100\,ns \approx 5\mu s$$

A operação completa:

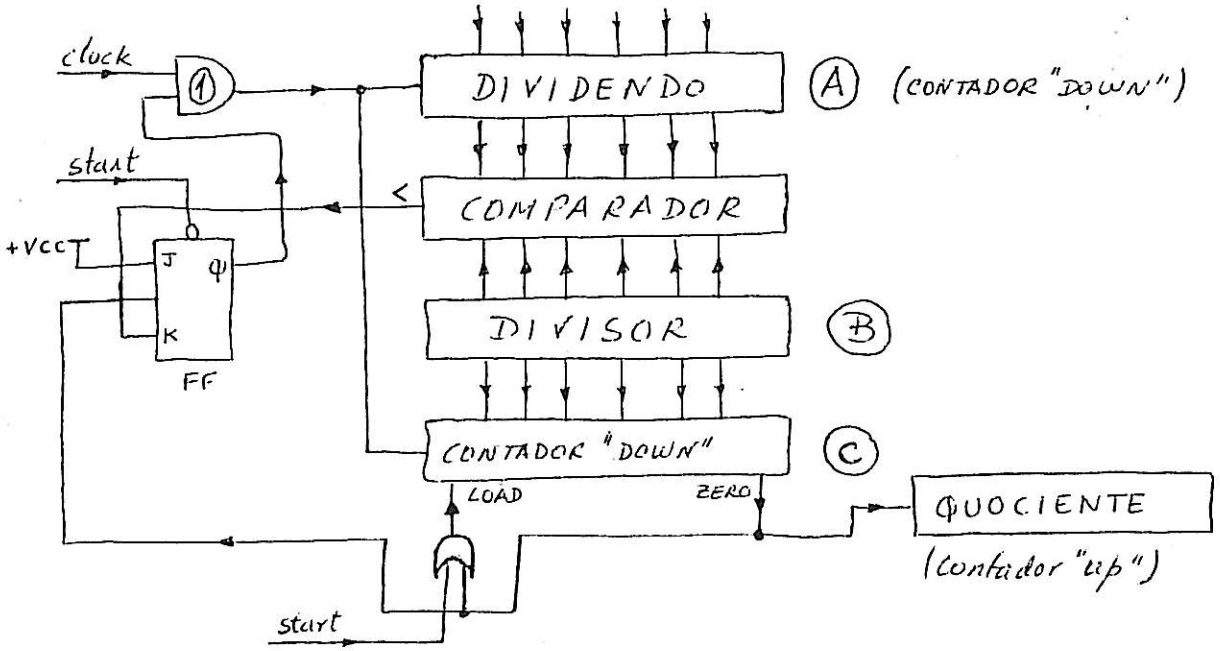
$$5\mu s \times 4 = 20\mu s$$

Reduzimos portanto o tempo de execução em cerca de 20 vezes. Para numeros maiores e operações elementares pequenas a redução é bastante mais drástica.

2.6.2. Divisão binária

Circuitos baseados em filosofias de operação semelhantes, podem ser desenvolvidos para executar divisões.

Apresentaremos apenas o circuito base, com os inconvenientes conhecidos, alertando no entanto para que, na prática estes circuitos têm que ser melhorados mediante sofisticações similares às descritas para os multiplicadores, se se quiserem tempos de execução razoáveis.



Operação. 1. Transferimos para o contador "down" Ⓐ e para o "buffer" Ⓑ respectivamente o dividendo e o divisor.

Colocamos o contador "up" do quociente no estado <u>zero</u>.

2. Ao ser dado o impulso de <u>start</u>, (que admitimos por questões de simplificação do esquema, ser suficientemente rápido e estar sincronizado com o sinal de "clock", de tal modo que acabe bastante antes do próximo impulso do "clock");

a- colocamos (por actuação no estado LOAD) o contador Ⓒ no estado que representa o divisor.

b- Colocamos o FF no estado $Q=1$, o que abre a gate ①, abrindo portanto caminho à passagem dos impulsos do "clock". Estes impulsos vão actuar no contador "DOWN" que contém o valor do Dividendo (A) e no outro contador "DOWN" (C) que contém o valor do divisor.

3. Quando este contador C atinge o valor <u>zero</u>, teremos subtraído ao valor do Dividendo, uma quantidade igual ao Divisor.

Então, neste instante, vamos dar um impulso que, por um lado vai ser acumulado no contador do quociente, e por outro lado vai de novo carregar o contador "DOWN" C, com o valor do divisor.

4. Chegados a este ponto, o processo continua a desenvolver-se de um modo semelhante a 3. até que o comparador dê indicação que no contador do dividendo, se encontra um número menor que o divisor.

Como este sinal (que vai a HIGH), vai actuar a entrada K do FF tipo JK, a operação vai continuar até ao fim da subtracção em curso nesse momento, após o que, o sinal de zero do contador C, além de dar um impulso para o contador do quociente, vai pôr o FF no estado Q=0, o que dá por terminada a operação.

O valor que se encontra no contador A representa o resto da divisão, encontrando-se o quociente no registo do mesmo nome.

2.7. Outras operações aritméticas

Além das quatro operações básicas mencionadas, as unidades aritméticas executam várias outras operações algébricas.

Estas operações, tais como raízes quadradas, logaritmos e funções trigonométricas, podem ser executadas com circuitos lógicos do tipo dos estudados até aqui, usando os algoritmos apropriados.

Muitas vezes, estes algoritmos são processos iterativos - em que um passo ou sequência de passos, é repetidamente efectuado até que a precisão desejada no resultado, seja obtida.

3. Circuitos Aritméticos integrados

Cada um dos elementos básicos dos circuitos descritos, é na maior parte das vezes, um único circuito integrado.

O "full-adder" básico, por exemplo, existe na forma de

quatro circuitos destes num só pacote integrado, ou então sob a forma de um adicionador de 4 bits, que portanto permite a soma de dois numeros de 4 bits de comprimento.

Full-adder's são por vezes combinados numa só "pastilha" com outros elementos lógicos, de modo a formarem adicionadores/subtractores, multiplicadores e unidades de lógica aritmética de uso geral.

Estas unidades de lógica aritmética (ALU's), têm a capacidade de executar qualquer uma das várias operações aritméticas ou lógicas. Estes pacotes de circuitos integrados, mediante o recurso a técnicas de "Large-scale-integration" (LSI), têm incluidos o equivalente a 80/100 gates lógicas.

Recentemente e dentro desta linha de desenvolvimento tecnológico, a integração dentro de uma só "pastilha" de um grande numero de circuitos lógicos, permitiu o aparecimento do microprocessador.

Estas "pastilhas", (LSI), contêm toda a lógica necessária à execução, não só de todas as funções da unidade aritmética e lógica de um computador, como todas as outras que são executadas numa unidade de processamento central.

Esta pastilha, associada com algumas outras que executem as funções de controle, input/output e memória, constituem um pequeno computador autónomo.

A evolução tecnológica do momento, encaminha-se para a muito curto prazo, ser comercializada, uma pastilha apenas, com todas as funções de um pequeno computador (CPU, IN/OUT, e MEMÓRIA).