P. SALENBAUCH* e E.A. SCHMITZ**

SUMÁRIO

Este artigo examina as relações da área ocupada e tempo de operação para alguns tipos de somadores binários. Discute-se, entre outra, os tipos: "ripple-carry", "previsão de carry" e "previsão de carry logarítmico". Para este último apresenta-se um programa para gerar automaticamente seu lay-out.

ABSTRACT

This paper examines space and time bounds for some binary adders. It includes a discussion of: "ripple-carry", "carry-look-ahead" and "logarithmic look-ahead adders". A routine for automatic lay-out generation of logarithimic look-ahead adders is presented.

* Engenheiro eletrônico (ITA, 1969), mestrado em Engenharia de Sistemas e Computação, COPPE - UFRJ, 1972; Compiladores, Linguagens de Programação, Sistemas Operacionais;

** Engenheiro eletrônico (UFRGS, 1971), M.Sc. COPPE - UFRJ, 1973; Ph.D. Imperial College 1980; Projeto de Circuitos Integrados.

Endereço de ambos os autores: Núcleo de Computação Eletrônica, Universidade Federal do Rio de Janeiro, CCMN, Cidade Universitária - Ilha do Fundão, Caixa Postal 2324 , CEP 20001, Rio de Janeiro, telefone (021) 290-3212, ramal 290.

O objetivo deste artigo é o estudo de alguns tipos somadores binários de N bits adequados para integração em grande escala. Ao estudar um determinado projeto de somador, levaremos em conta não somente o tempo de soma e o espaço ocupado pelo somador, como também a regularidade de cada módulo e o problema topológico de ligação entre estes diversos módulos.

Na seção 2, lembramos que teoricamente uma soma de operandos de qualquer comprimento pode ser feito em tempo constante, desde que tenhamos o circuito adequado - cujo espaço cresce exponencialmente com o número de bits. A seguir, focalizamos a situação inversa , em que mostramos que com um circuito fixo podemos somar operandos de qualquer comprimento com tempo crescendo linearmente com o número de bits dos operandos.

A seção 3 apresenta o "Ripple Carry", que é um dos esquemas mais simples de realizar a soma, tendo tanto o espaço como o tempo O(N). A seguir é mostrado o "Manchester Carry - Chain", um aperfeiçoamento do "Ripple-Carry", com algumas características próprias.

Na seção 4 é apresentado o método de "Carry Lookahead". A versão logarítmica do somador é discutida e um programa para geração é apresentado.

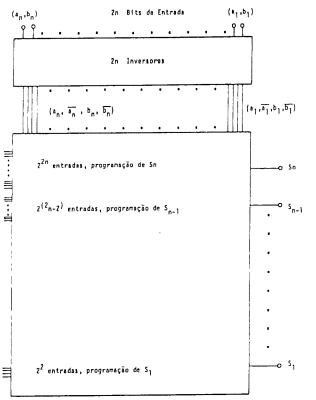
Finalmente, a seção 5 apresenta comparações entre os vários métodos e as conclusões finais.

COTA INFERIOR DE TEMPO E ESPAÇO

COTA INFERIOR DE TEMPO

Como sabemos, qualquer função lógica pode ser realizada em 3 etapas: na primeira, obtemos os inversos dos dados de entrada (se eventualmente estes já estão disponíveis, esta etapa pode ser dispensada); na segunda, obtemos os P-termos utilizando a operação lógica "E"; finalmente na terceira etapa obtemos a função desejada fazendo o "OU" dos P-termos obtidos na segunda etapa.

Como a soma binária de N bits nada mais é do que N funções de 2N, 2N-2, 2N-4, ..., 2 variáveis binárias, podemos utilizar esta idéia para obter um somador, como no esquema dado a seguir:



Esta é a maneira mais rápida de achar a soma, cujo tempo de operação é independente da largura dos operandos. Este tempo tem o valor da soma do tempo de um inversor mais o tempo do "E" mais o tempo do "OU". Infelizmente, no entanto, a área necessária é exponencial (basta ver que apenas para obtermos o s_n precisamos de (2 ** (2*N)) linhas).

O esquema acima apresentado supõe portas lógicas sem restrição de "fan-in". Se considerarmos que as portas podem ter "fan-in" máximo igual a 'R', então a cota inferior de tempo Winograd será:

TEMPO = O(LOG(R) (2+N))

Porém tendo uma função área:

A = O(2**N)

Brent mostrou ser possível construir somadores com "fan-in" limitado de forma que:

T = O((1+E) *LOG(R) N)

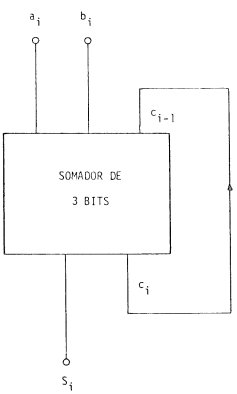
A = O(N*LOG N)

Onde E → Ø quando N → Infinito. (Brent)

COTA INFERIOR DE ESPAÇO

Tomando o extremo oposto do caso anterior, podemos também realizar a soma de N bits tendo apenas um somador de 3 bits (isto é, um circuito de complexidade constante). Com isto sacrificaremos o tempo de soma, obtendo somas em um tempo linearmente dependente da largura dos operandos.

Seria um esquema do seguinte tipo:



Este esquema, com apenas um somador, o carry de cada dígito é realimentado para realizar a soma dos bits de ordem imediatamente superior no ciclo seguinte da soma.

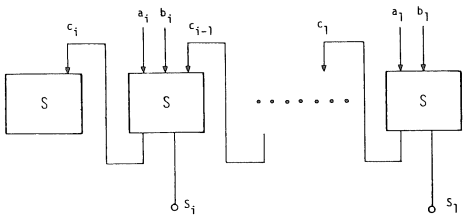
Neste caso temos: espaço O(N), tempo O(N) e espaço x tempo $O(N^2)$. Este esquema é o que provavelmente tem o melhor valor (espaço x tempo).

SOMADOR COM RIPPLE-CARRY

No caso simples a soma de N bits é dividida em N somadores de 3 bits, cada um calculando:

$$S_i = A_i \oplus B_i \oplus C_{i-1}$$

$$C_i = A_iB_i + (A_i + B_i).C_{i-1}$$



Donde:

ÁREA = O(N)

E:

TEMPO = O(3*N)

Pois em cada módulo acima, para se obter a soma a partir dos 3 sinais de entrada, necessitamos de 3 níveis lógicos.

MANCHESTER CARRY CHAIN

É uma variante do somador com ripple carry onde é colocado um meio mais rápido para propagação do carry. Ao invés do carry C_{i-1} ser processado como uma variável de entrada do bloco somador (gastando no mínimo 2 unidades de atraso para se propagar ao bloco seguinte) faz-se com que o carry seja gerado ou bloqueado em cada estágio por meio de chaves colocadas em série com o "percurso de carry". Esta operação de acionamento de chaves pode ser feita simultaneamente em todos os estágios de forma que o tempo de soma fica sendo dependente do comprimento físico da linha do carry.

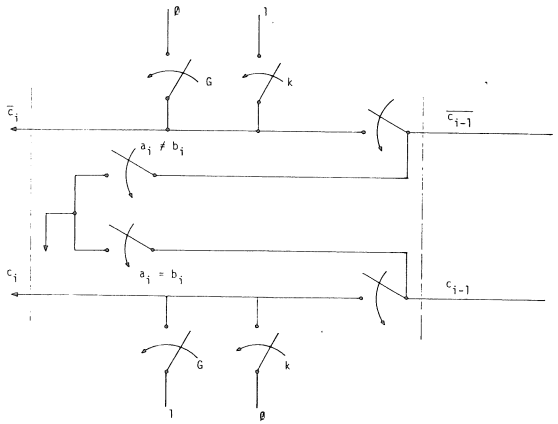
O diagrama a seguir ilustra a idéia.

Onde:

$P = A_i \oplus B_i$: Carry é propagado se $A_i \neq B_i$

$K = (-A_i).(-B_i)$: Carry de saída será zero ("Kill")

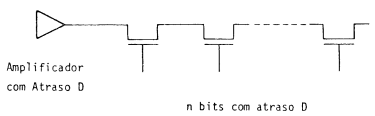
$G = A_i.B_i$: Carry de saída será um ("Gerador").



A área ocupada é O(N), porém temos dois casos a analisar para o tempo de propagação:
Caso 1 - Supondo que a propagação do carry seja em uma linha de transmissão com parâmetros distribuídos. No pior caso o tempo de propagação é A*L**2, onde L é o comprimento da linha.

T = O(L**2)

Caso 2 - Supondo que amplificadores sejam colocados a intervalos convenientes da linha de carry (Mead & Conway 80).



Temos (N/M) seções com atrazo 4*D. Escolhendo-se M de tal forma que o atrazo D seja igual ao de um nível de lógica:

TEMPO = (N/M)*4 = O(4*N/M)
COMO EM GERAL M = 4,
TEMPO = O(N)

SOMADOR COM PREVISÃO DE "CARRY"

PREVISÃO DE "CARRY"

Ao se observar o circuito de um somador notamos que o que determina basicamente o tempo

de soma é a espera de cada somador elementar pelo carry do estágio anterior.

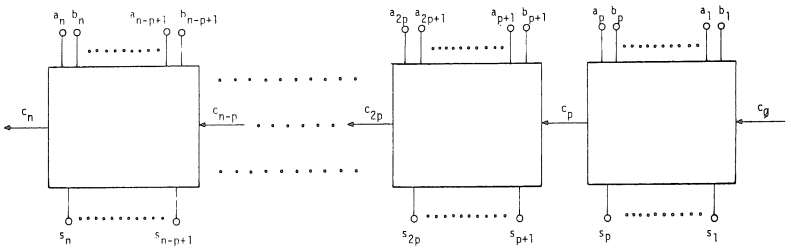
Uma idéia alternativa consiste em dividir os operandos de N bits, em grupos de P bits (vamos supor que N é múltiplo de P). Se pudéssemos saber instantaneamente qual seria o carry resultante de uma soma de um grupo de P bits, então seria possível saber-se o resultado da soma dos N bits com uma velocidade P vezes maior.

Na realidade, este tempo de previsão não é nulo. Vamos supor que o tempo de previsão de carry de um grupo de P bits é T. Neste caso o tempo de soma dos N bits será:

N/P*T * (Tempo de um Estágio)

Podemos notar que este esquema é realmente mais rápido, por um fator de P vezes, mas a sua velocidade continua sendo uma função linear de N.

Esta é a idéia central do somador de "Carry Lookahead", que tem um esquema do seguinte tipo:



Cada caixa acima tem um circuito convencional (ripple-carry) para obter a soma dos P dígitos e mais um circuito especial para obter o carry de saída (antes da soma).

Para este circuito teremos: espaço O(N), tempo O(N/P), espaço x tempo O(N**2/P).

Nos somadores de "Carry Lookahead" existem duas funções auxiliares de grande importância, que são definidos como:

$$G_i = A_i . B_i$$
$$P_i = A_i \oplus B_i$$

O G_i é a função de "Geração de Carry", e informa se no i-ésimo estágio será ou não gerado um carry. O P_i é a função de "Propagação de Carry", e informa se o i-ésimo estágio propagará ou não um carry que venha do estágio I-1.

Como sabemos que:

$$S_i = A_i \oplus B_i \oplus C_{i-1}$$
$$C_i = A_i . B_i + A_i . C_{i-1} + B_i . C_{i-1}$$

Podemos deduzir que:

$$S_i = P_i \oplus C_{i-1}$$
$$C_i = G_i + P_i . C_{i-1}$$

A segunda expressão é bem intuitiva, informando que este estágio irá ter um carry na saí

da, se ele for gerado aqui ou propagado do estágio anterior.

A vantagem de usar estas funções auxiliares P e G é que teremos circuitos mais simples do que se calculássemos os carry's diretamente dos A_i e B_i. Assim, para um bloco de 4 bits (P = 4), teremos as seguintes funções para o carry de saída:

$$C_4 = G_4 + G_3.P_4 + G_2.P_4 + G_1.P_2.P_3.P_4 + C_\emptyset.P_1.P_2.P_3.P_4.$$

Esta fórmula é também bastante intuitiva, e significa que ou o carry é gerado no quarto estágio, ou então gerado no terceiro e propagado no quarto, e assim por diante.

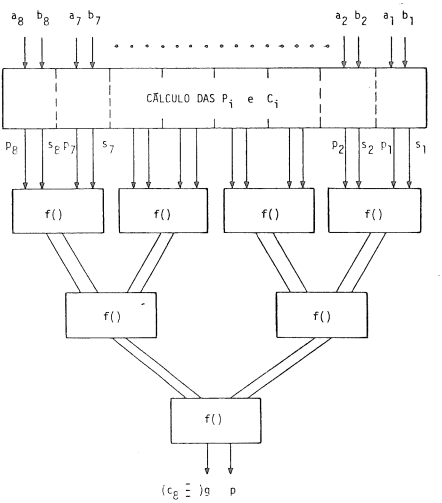
CARRY LOOKAHEAD LOGARÍTMICO

Uma outra solução, mais rápida e sofisticada, utiliza uma função auxiliar mais complexa:

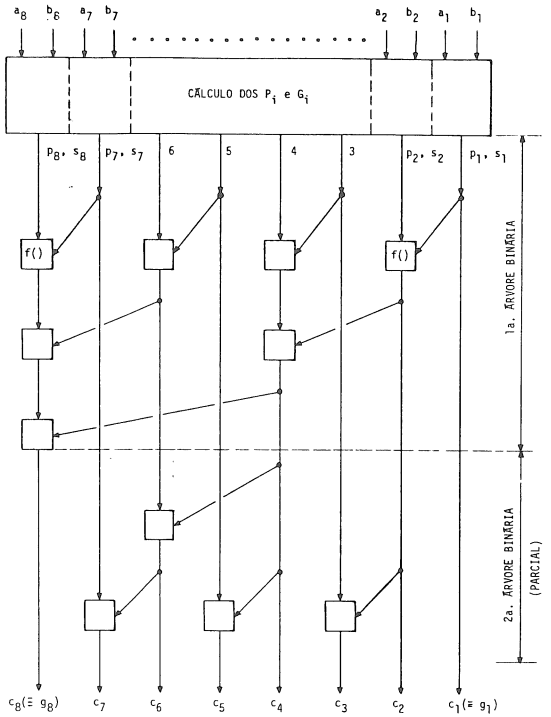
$$F((G_i, P_i), (G_j, P_j)) = (G_i + P_i.G_j, P_i.P_j).$$

Onde I e J são quaisquer (não necessariamente adjacentes). É possível mostrar que esta função pode ser usada para calcular todos os carry's e que ela é associativa.

Isto tem a vantagem de que estes cálculos podem ser iniciados em todos os estágios em paralelo, dando um circuito em forma de árvore binária:



Este circuito produz, em um tempo logarítmico, o carry resultante da soma dos N bits (no caso N = 8). No entanto, para obtermos os 8 bits da soma, não basta termos C_8. Necessitamos também ter C_1, C_2, ..., C_7. Para obtermos estes carry's restantes, precisamos de mais uma árvore binária (parcial), montado segundo o esquema a seguir:



Baseados nesta idéia, primeiramente descrita por Brent e Kung (79), podemos construir um somador, cujas características de tempo e espaço dadas por:

ESPAÇO = O(n.log(n))

TEMPO = O(log(n))

Para que o lay-out se torne regular Brent introduziu a noção dos "processadores brancos" e dos "processadores pretos" cujas funções lógicas são dadas por:

Processador preto: recebe como entrada - p_i, $\hat{p}_i$, g_i, $\hat{g}_i$

gerando: pout = $p_i p_j$

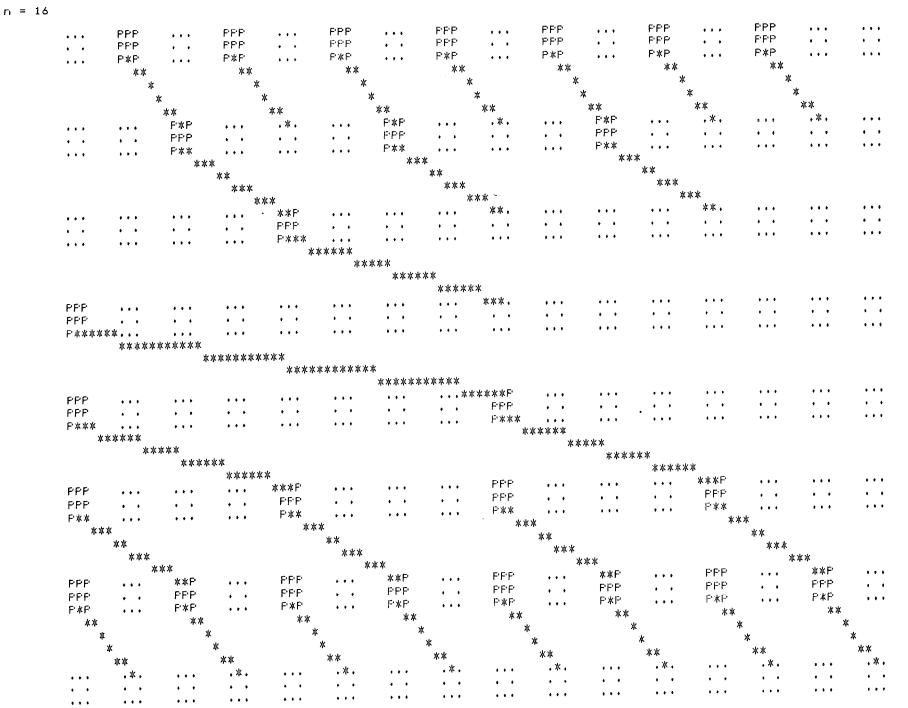
gout = $g_i + p_i g_j$

Processador branco: recebe como entrada (gin, pin) e gera dois grupos de sinais iguais tais que, gout = g_i e pout = p_i.

Finalmente, para a geração de soma basta acrescentar um estágio final que realize:

$S_i = P_i + C_{i-1}$.

Como o lay-out obtido é extremamente regular, temos no apêndice 1 o texto fonte de um programa para geração automática de somadores logarítmicos. Para adaptá-lo a uma dada tecnologia, basta se definir os lay-outs (em CIF, p.ex.) dos processadores brancos, pretos e de sua interconexão. A fig. abaixo mostra uma saída simbólica do programa para um somador de 16 bits.



CONCLUSÕES

Para somadores de tamanho moderado, a melhor opção continua sendo o Manchester Carry Chain, que tem tanto a área como o tempo linear com n.

Para somadores de tamanho grande, onde a velocidade é um fator importante, o circuito com previsão de carry logarítmico é o mais indicado, pois com ele temos um lay-out regular, adaptado para circuito com alta integração com uma penalidade de área que cresce

com n.log(n). A tabela abaixo mostra um resumo dos resultados obtidos.

SOMADOR	TEMPO	ESPAÇO
Ripple-Carry	O(3n)	O(n)
Manchester	O(n)	O(n)
Lookahead simples	$O(\frac{3}{4}\ n)$	O(n)
Lookahead logarítmico	O(2 log(n))	O(n.log(n))

BIBLIOGRAFIA

WINOGRAD, S. "On the time required to perform addition", J.A.C.M., V2-2, Abril 1965;

BRENT, R. "On the addition of binary members", Trans. on Computers, C-19, Agosto 1970;

MEAD C.A., Conway, L. "An introduction of VLSI systems", Addison-Wesley, 1980;

HWANG K. "Computer Arithmetic", John Wiley, 1979;

BRENT R., KUNG H. "A regular Lay-out for parallel adders", Technical Report, CMU-CS-79-131, Carnegie-Melon Univ., 1979.

```
/*
 *      Programa para gerar automaticamente
 *      o Layout do Somador Lookahead
 *
 *      Modo de usar: somador l
 *        onde l = log(2) n
 */

short   rx, ry;          /* Coordenadas */

short   l, n;            /* n = nr. de Bits, l = log(2) (n) */

main (argc, argv)
char **argv;
{
        register i;

        if ((l = atoi (argv[1])) <= 0) {        /* le o valor de l */
                printf ("l invalido\n");
                exit (1);
        }

        n = 1 << l;                             /* calcula n */

        for (i = 0; i <= l; i++) {              /* Linhas da Primeira Arvore */
                seek (0, i, 0);
                linha1 (1 << i);
        }

        for (i = 1; i < l; i++) {               /* Linhas da Segunda Arvore */
                seek (0, i+1, 0);
                linha2 (1 << (l-i) );
        }

        exit (0);

}       /* end main */

linha1 (i)
register i;
{
        register j;

        for (j = 0; j < n; j++) {
                if (ry == 0)
                        branco ();
                else if (j % i == 0) {
                        preto ();
                        fio (rx + (i >> 1), ry-1);
                } else
                        branco ();
                seek (1, 0, 1);
        }

}       /* end linha1 */

linha2 (i)
register i;
{
        register j;

        for (j = 0; j < n; j++) {
                if (j % i == (i >> 1) && j < n - i) {
                        preto ();
```

```
                        fio (rx+ (i>>1), ry-1);
                } else
                        branco ();

                seek (1, 0, 1);

        }

}       /* end linha2 */


fio (x, y)
{
        /*************************************************************
        *                                                            *
        *       Neste Ponto Insere-se o codigo para gerar uma        *
        *       ligacao do ponto (x,y) ao (rx,ry)                    *
        *                                                            *
        *************************************************************/

}       /* end fio */

seek (x, y, r)
{
        if (r) {
                rx += x;
                ry += y;
        } else {
                rx = x;
                ry = y;
        }

}       /* end seek */

branco ()
{
        /*************************************************************
        *                                                            *
        *       Neste ponto insere-se o Codigo para gerar            *
        *       um Processador Branco no ponto (rx,ry)               *
        *                                                            *
        *************************************************************/

}       /* end branco */

preto ()
{

        /*************************************************************
        *                                                            *
        *       Neste ponto insere-se o Codigo para gerar            *
        *       um Processador Preto no ponto (rx,ry)                *
        *                                                            *
        *************************************************************/

}       /* end Preto */

#SYS#
```