Fundamentos de Arquitetura de Computadores

Tiago Alves

Faculdade UnB Gama
Universidade de Brasília

Sumário

Módulo 04

- Somadores
- Multiplicadores
- ULA

A adição é a operação aritmética mais comumente realizada em circuitos.

Um circuito **somador** (*adder*) combina dois operandos aritméticos usando as regras da adição binária. Vimos que o mesmo circuito pode ser usado para somar palavras sem sinal (*unsigned*) e em complemento-de-2.

O circuito para somar dois bits (ou somar duas palavras de 1 bit) é chamado de **somador parcial** (*half adder*). Este circuito soma dois bits, A e B, e gera uma soma de dois bits (de 0 a 2).

A	**B**	**C**	**S**
0	**0**	**0**	**0**
0	**1**	**0**	**1**
1	**0**	**0**	**1**
1	**1**	**1**	**0**

O que gera as equações da soma parcial: ...

O circuito para somar dois bits (ou somar duas palavras de 1 bit) é chamado de **somador parcial** (*half adder*). Este circuito soma dois bits, A e B, e gera uma soma de dois bits (de 0 a 2).

A	**B**	**C**	**S**
0	**0**	**0**	**0**
0	**1**	**0**	**1**
1	**0**	**0**	**1**
1	**1**	**1**	**0**

O que gera as equações da soma parcial:

$$\mathbf{C = A \cdot B}$$
$$\mathbf{S = A \oplus B}$$

Somador Total

O circuito para somar três bits (ou somar duas palavras de 1 bit mais um *carry*) é chamado de **somador total** (*full adder*). Este circuito soma três bits, A, B e C_{in}, e gera uma soma de dois bits (de 0 a 3).

$\mathbf{C_{in}}$	**A**	**B**	$\mathbf{C_{out}}$	**S**
0	**0**	**0**	**0**	**0**
0	**0**	**1**	**0**	**1**
0	**1**	**0**	**0**	**1**
0	**1**	**1**	**1**	**0**
1	**0**	**0**	**0**	**1**
1	**0**	**1**	**1**	**0**
1	**1**	**0**	**1**	**0**
1	**1**	**1**	**1**	**1**

O que gera as equações da soma total ...

Somador Total

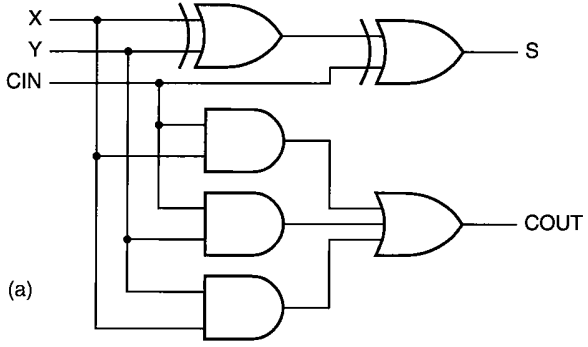
O circuito para somar três bits (ou somar duas palavras de 1 bit mais um *carry*) é chamado de **somador total** (*full adder*). Este circuito soma três bits, A, B e C_{in}, e gera uma soma de dois bits (de 0 a 3).

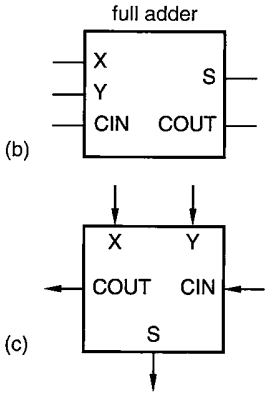
$\mathbf{C_{in}}$	**A**	**B**	$\mathbf{C_{out}}$	**S**
0	**0**	**0**	**0**	**0**
0	**0**	**1**	**0**	**1**
0	**1**	**0**	**0**	**1**
0	**1**	**1**	**1**	**0**
1	**0**	**0**	**0**	**1**
1	**0**	**1**	**1**	**0**
1	**1**	**0**	**1**	**0**
1	**1**	**1**	**1**	**1**

O que gera as equações da soma total:

$$\mathbf{C_{out}} = \mathbf{A} \cdot \mathbf{B} + \mathbf{A} \cdot \mathbf{C_{in}} + \mathbf{B} \cdot \mathbf{C_{in}}$$
$$\mathbf{S} = \mathbf{A} \oplus \mathbf{B} \oplus \mathbf{C_{in}}$$

Somador Total





Somadores de n bits

Considere um circuito somador de duas palavras de n bits. Como este é um circuito puramente combinacional, podemos montar sua tabela verdade (com 2^{2n} linhas ou entradas possíveis), e minimizá-lo usando qualquer técnica que desejarmos.

Porém, para somar apenas **palavras de 4 bits**, temos 8 variáveis (9, se considerarmos o *carry in* da entrada) e 256 linhas (512 com o *carry*) na tabela verdade.

Uma outra opção é utilizar um circuito **iterativo**! Note que, durante a soma, fazemos a mesma operação em cada bit.

Dividir para conquistar!

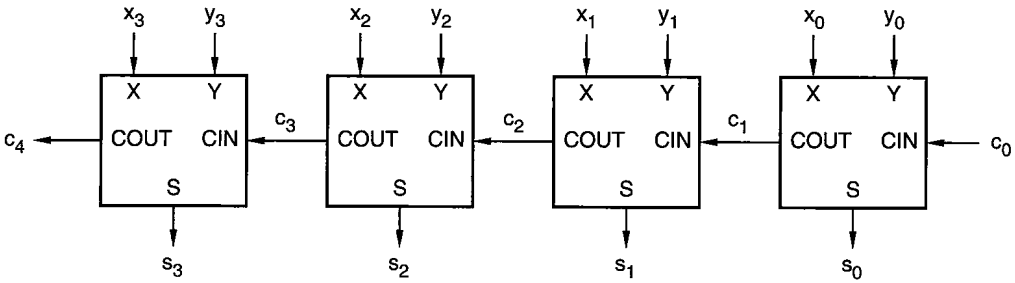
UnB Gama

A adição binária pode então ser feita usando o algoritmo:

1. Ajuste $C_0 = \mathbf{0}$ e $i = 0$.
2. Some os bits C_i, A_i e B_i para obter S_i (saída primária) e C_{out} (saída de cascateamento).
3. Incremente i.
4. Se $i < n$, volte ao passo 2.

Assim, precisamos de um módulo que realize a soma entre C_i, A_i e B_i para obter S_i e C_{out}. Bom, mas eu já conheço um módulo digital capaz de realizar essa função: o circuito **somador total**.

O circuito de somador iterativo (agrupando somadores totais) é chamado de **Ripple Adder**.



Podemos construir um circuito "subtrator" binário análogo ao somador usando a tabela verdade da subtração com *borrow*, que realiza a subtração $A - B$.

$\mathbf{B_{in}}$	**A**	**B**	$\mathbf{B_{out}}$	**D**
0	**0**	**0**	**0**	**0**
0	**0**	**1**	**1**	**1**
0	**1**	**0**	**0**	**1**
0	**1**	**1**	**0**	**0**
1	**0**	**0**	**1**	**1**
1	**0**	**1**	**1**	**0**
1	**1**	**0**	**0**	**0**
1	**1**	**1**	**1**	**1**

O que gera as equações da subtração total ...

Subtrator

Podemos construir um circuito "subtrator" binário análogo ao somador usando a tabela verdade da subtração com *borrow*, que realiza a subtração $A - B$.

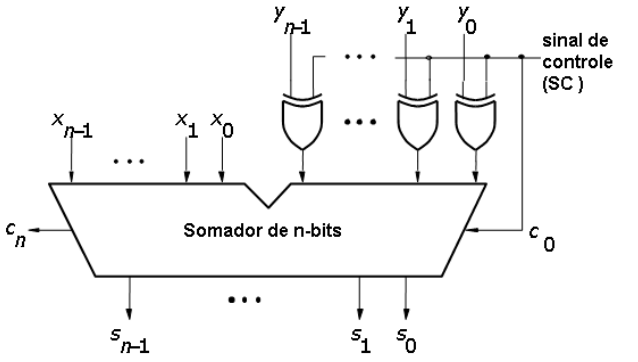
$\mathbf{B_{in}}$	$\mathbf{A}$	$\mathbf{B}$	$\mathbf{B_{out}}$	$\mathbf{D}$
$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$
$\mathbf{0}$	$\mathbf{0}$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$
$\mathbf{0}$	$\mathbf{1}$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{1}$
$\mathbf{0}$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{0}$	$\mathbf{0}$
$\mathbf{1}$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{1}$	$\mathbf{1}$
$\mathbf{1}$	$\mathbf{0}$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{0}$
$\mathbf{1}$	$\mathbf{1}$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$
$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$

O que gera as equações da subtração total:

$$\mathbf{B_{out}} = \overline{\mathbf{A}} \cdot \mathbf{B} + \overline{\mathbf{A}} \cdot \mathbf{B_{in}} + \mathbf{B} \cdot \mathbf{B_{in}}$$
$$\mathbf{D} = \mathbf{A} \oplus \mathbf{B} \oplus \mathbf{C_{in}}$$

Subtração usando o Ripple Adder

Porém, vimos que, para subtrair números em complemento de 2, basta somar o número com o seu complemento (isto é, ao invés de fazer $A - B$, fazemos $A + (-B)$). Para obter o inverso de um número em complemento de 2, basta inverter todos os bits e somar 1. Logo, podemos fazer o circuito:



Ripple Adder

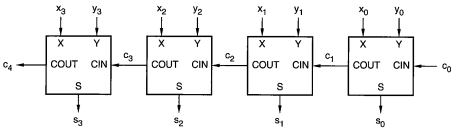
O maior problema do *ripple adder* é o **atraso**.

No pior caso, o *carry* tem que ser propagado do bit menos significativo até a saída *carry out* do módulo mais significativo (por exemplo, quando somamos $\mathbf{11}...\mathbf{11}$ com $\mathbf{00}...\mathbf{01}$). O atraso do pior caso é:

$$T_{ADD} = T_{ABCout} + (n-2)\,T_{CinCout} + T_{TCinS}$$

onde:

- T_{ADD}: tempo total necessário para consolidar a soma das duas palavras;
- T_{XYCout}: atraso na geração de Cout demandado pelo estágio menos significativo;
- $T_{CinCout}$: atraso entre Cin e Cout nos estágios intermediários;
- T_{TCinS}: atraso entre Cin a geração de S no último estágio.



Podemos criar uma lógica de soma com apenas 2 níveis de atraso (típico da estrutura AND-OR) se tentarmos minimizar o circuito diretamente da tabela verdade, ou seja, fazendo as equações para cada bit de saída s_i a partir das entradas ($a_i...a_0, b_i...b_0$ e c_{in}).

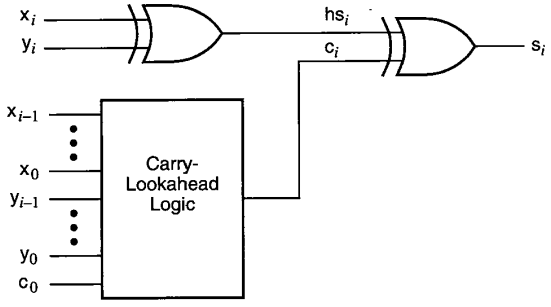
Porém, a partir de s_2 (ou seja, apenas o terceiro bit da soma!), as equações já ficam muito grandes e necessitariam de 14 4-input ANDs, 4 5-input ANDs e uma 18-input OR! E isso apenas para s_2.

Carry Lookahead

Podemos acelerar este atraso olhando as equações:

$$s_i = a_i \oplus bi \oplus c_i$$

Já vimos que tentar abrir completamente esses termos gera muitas complicações. A ideia do circuito com *carry lookahead* é manter as portas **xor** e gerar apenas o *carry*.



Para isso, ele usa duas definições chave. Para uma combinação a_i,b_i no estágio i, dizemos que:

- um *carry* é **gerado** nesse estágio se ele produz um *carry out* $\mathbf{1}$ **independente** das entradas $a_{i-1}...a_0$, $b_{i-1}...b_0$ e c_{in}.
- um *carry* é **propagado** nesse estágio se ele produz um *carry out* $\mathbf{1}$ se e somente se o *carry in* desse estágio for $\mathbf{1}$.

Ou seja, temos:

$$g_i = a_i \cdot b_i$$
$$p_i = a_i + b_i$$

Ou seja, um estágio **gera** um *carry out* incondicionalmente se ambas as entradas desse estágio são $\mathbf{1}$ e ele **propaga** um *carry in* se pelo menos um de seus adendos for $\mathbf{1}$. Assim, podemos escrever o *carry out* como:

$$c_{i+1} = g_i + p_i \cdot c_i$$

Assim:

$$
\begin{aligned}
c_1 &= g_0 + p_0 \cdot c_0 \\
c_2 &= g_1 + p_1 \cdot c_1 \\
&= g_1 + p_1 \cdot (g_0 + p_0 \cdot c_0) \\
&= g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0 \\
c_3 &= g_2 + p_2 \cdot c_2 \\
&= g_2 + p_2 \cdot (g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0) \\
&= g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0 \\
c_4 &= g_3 + p_3 \cdot c_3 \\
&= g_3 + p_3 \cdot (g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0) \\
&= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0
\end{aligned}
$$

Cada uma dessas equações corresponde a um circuito com 3 níveis de atraso (um para gerar g_i e p_i) e dois para a estrutura AND-OR. (Análise em pior caso!)

Multiplicação em binário

A multiplicação em binário se dá da mesma forma que a multiplicação em decimal: adicionando uma lista de multiplicandos deslocados computados de acordo com os dígitos do multiplicador.

Na multiplicação em binário esse processo é ainda mais fácil, pois ou o bit do multiplicador é 0 (e o multiplicando deslocado é 0) ou é 1 (e o multiplicando deslocado é igual ao multiplicando).

$$\begin{array}{r} 11 \\ \times\ 13 \\ \hline 33 \\ 11 \\ \hline 143 \end{array}$$

$$\begin{array}{rl} 1011 & \text{multiplicando} \\ \times\ 1101 & \text{multiplicador} \\ \hline 1011 & \\ 0000 & \text{multiplicando} \\ 1011 & \text{deslocados} \\ 1011 & \\ \hline 10001111 & \text{produto} \end{array}$$

Embora possamos projetar circuitos *sequenciais* para realizar a multiplicação, não há nada inerentemente sequencial ou dependente do tempo na multiplicação binária.

Podemos expressar a multiplicação de dois operandos X e Y de n bits em uma tabela verdade (com 2^{2n} linhas) e elaborar um circuito AND-OR para realizar essa multiplicação.

O problema é que, novamente, essa tabela verdade cresce muito rapidamente.

Multiplicadores de 3 bits

Considere a multiplicação de dois operandos de 3 bits utilizando o algoritmo de "deslocar e acumular". Para representar o resultado sem que haja *overflow* precisamos de $2n = 6$ bits (o maior número inteiro se sinal que pode ser representado é $7 \times 7 = 49$).

Note que o produto entre dois bits é super simples: ele só é $\mathbf{1}$ se ambos os bits forem $\mathbf{1}$! Logo, o produto de dois bits pode ser feito utilizando uma simples porta AND.

$$
\begin{array}{cccccc}
 & & & x_2 & x_1 & x_0 \\
 & & \times & y_2 & y_1 & y_0 \\
\hline
 & & & y_0x_2 & y_0x_1 & y_0x_0 \\
+ & & y_1x_2 & y_1x_1 & y_1x_0 & \\
+ & y_2x_2 & y_2x_1 & y_2x_0 & & \\
\hline
p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$

$$
\begin{array}{cccccccc}
 & & & & & x_2 & x_1 & x_0 \\
 & & & & \times & y_2 & y_1 & y_0 \\
\hline
 & & & & & y_0x_2 & y_0x_1 & y_0x_0 \\
+ & & & & y_1x_2 & y_1x_1 & y_1x_0 & \\
+ & & & y_2x_2 & y_2x_1 & y_2x_0 & & \\
\hline
 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 &
\end{array}
$$

Para calcular o produto, basta somar as colunas:

$$
\begin{aligned}
p_0 &= y_0x_0 \\
p_1 &= y_0x_1 + y_1x_0 \\
p_2 &= y_0x_2 + y_1x_1 + y_2x_0 \\
p_3 &= y_1x_2 + y_2x_1 \\
p_4 &= y_2x_2 \\
p_5 &= ?
\end{aligned}
$$

Como definir a expressão para cálculo de p_5?

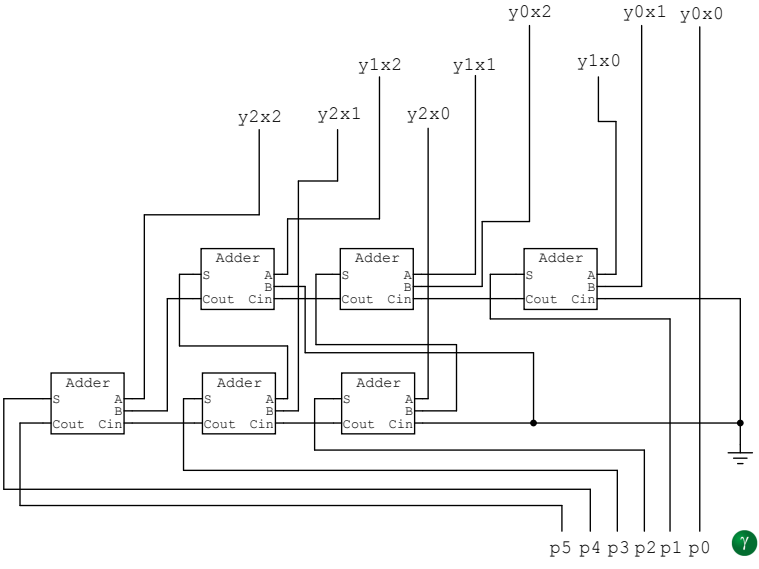
O que estamos esquecendo é que, assim como a multiplicação, a adição é binária e pode gerar um *carry out*!

$$
\begin{aligned}
p_0 &= y_0 x_0 \\
p_1 &= y_0 x_1 + y_1 x_0 \\
p_2 &= y_0 x_2 + y_1 x_1 + y_2 x_0 + c_1 \\
p_3 &= y_1 x_2 + y_2 x_1 + c_2 \\
p_4 &= y_2 x_2 + c_3 \\
p_5 &= c_4
\end{aligned}
$$

Podemos usar somadores completos para realizar essa soma. Porém, note que o termo p_2 soma quatro bits e, portanto, pode precisar ser propagado até o termo p_4.

Multiplicadores de 3 Bits

$$p_0 = y_0 x_0$$
$$p_1 = y_0 x_1 + y_1 x_0$$
$$p_2 = y_0 x_2 + y_1 x_1 + y_2 x_0 + c_1$$
$$p_3 = y_1 x_2 + y_2 x_1 + c_2$$
$$p_4 = y_2 x_2 + c_3$$
$$p_5 = c_4$$



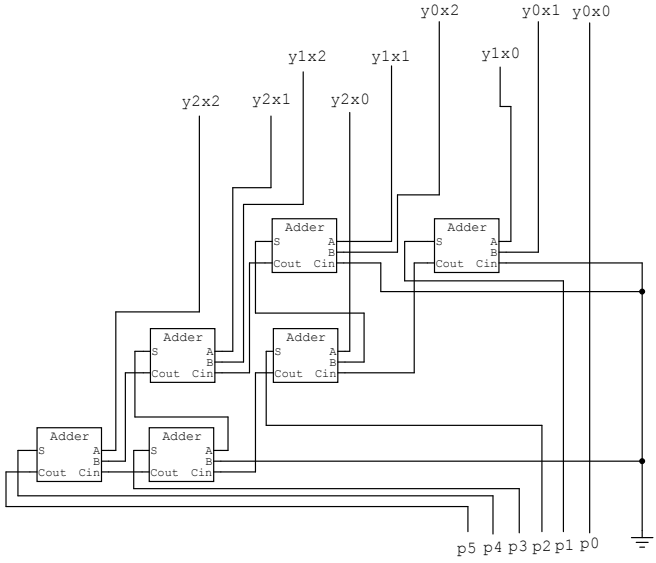
Obviamente, o maior problema do circuito anterior é o **atraso**.

O circuito tem apenas **seis** somadores mas, no pior caso, o sinal deve ser propagado por **cinco** deles até que chegue a saída.

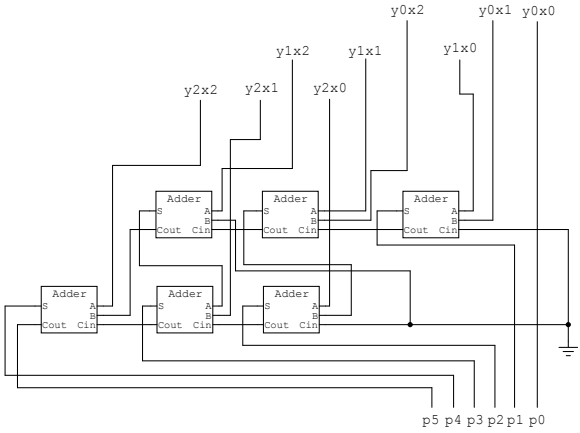
Podemos "acelerar" este este circuito utilizando a ideia de *carry save addition*, ligando o *carry out* de um somador no somador *abaixo* dele, não ao lado.

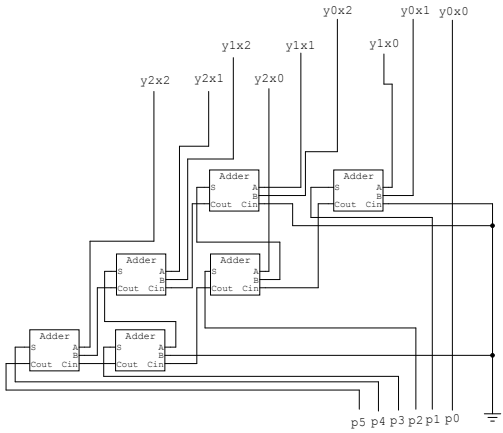
Multiplicadores de 3 Bits

$$
\begin{aligned}
p_0 &= y_0x_0 \\
p_1 &= y_0x_1 + y_1x_0 \\
p_2 &= y_0x_2 + y_1x_1 + y_2x_0 + c_1 \\
p_3 &= y_1x_2 + y_2x_1 + c_2 \\
p_4 &= y_2x_2 + c_3 \\
p_5 &= c_4
\end{aligned}
$$



Multiplicadores de 3 Bits



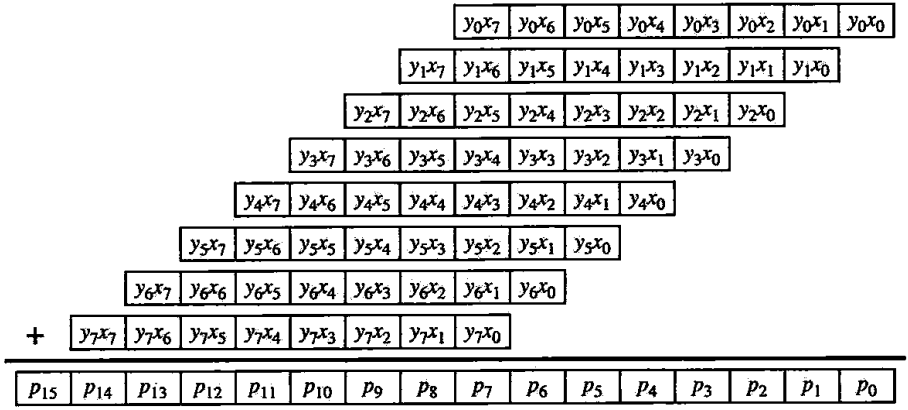


Multiplicadores de 3 Bits

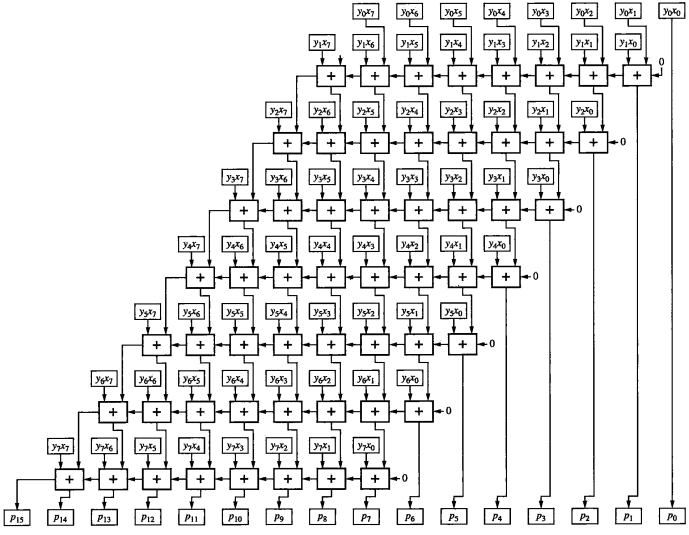
No multiplicador de apenas 3 bits, o *carry save addition* economiza apenas 1 atraso, pois o pior caso do circuito anterior é que um *carry* se propague por 4 somadores.

Pode não parecer muito, especialmente porque o entendimento do circuito fica um pouco mais complexo, mas em um circuito multiplicador de 8 bits a economia é de 20 somadores para 14 (e ainda menos se utilizarmos um circuito *carry lookahead* na última linha).

Multiplicadores de 8 Bits

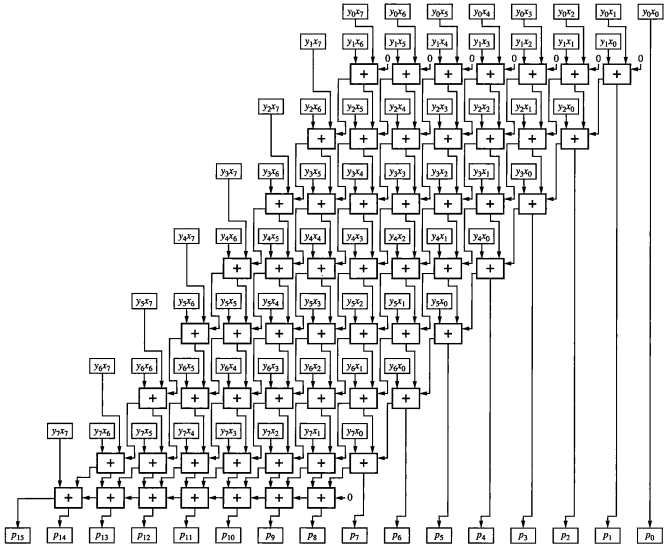


Multiplicadores de 8 Bits



Multiplicadores de 8 Bits

Versão rápida: equivalente ao carry-save.



Central Processing Unit - CPU

A CPU é o circuito eletrônico em um computador que executa **instruções** de um *programa*.

Essas instruções são compostas de operações classificadas em, basicamente, três categorias:

- entrada/saída;
- operações de controle e
- e operações básicas aritméticas e lógicas.

Para executar uma instrução (que poderá pertencere às três categorias anteriores), a CPU faz três passos:

1. **Fetch** (busca) - Busca na memória qual instrução deve ser executada.
2. **Decode** (decodifica) - Decodifica essa instrução, lendo o que deve ser feito e com quais operandos.
3. **Execute** (executa) - Efetivamente executa essa instrução.

Ao fim de uma instrução, a CPU atualiza o *contador de programa* e repete esses passos novamente.

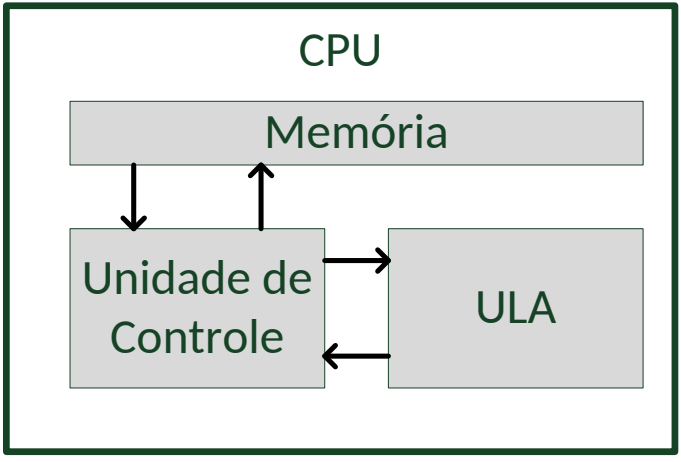
As instruções que uma CPU pode executar são (em geral) simples e *sempre* bem definidas.

Instrução		Descrição	Tipo
ADD	A,#20	Adiciona o valor 20 ao conteúdo posição de memória A	**Aritmética**
AND	AL,BL	Faz o AND bit a bit dos conteúdos dos registradores AL e BL. Resultado armazenado em AL.	**Lógica**
MOV	A,R0	Copia o conteúdo da posição R0 para a posição A	**Transferência de dados**
INC	R2	Incrementa a posição de memória R2	**Aritmética**
JMP	LB0	Pula para a instrução rotulada como LB0	**Controle**
JB	F0,LB1	Se a flag F0 for 1, pula para LB1	**Controle**
CJNE	R1,#32,LB2	Compara o valor da posição R1 com 32 e, se forem diferentes, pula para LB2	**Controle**

Note que muitas dessas instruções envolvem operações aritméticas (ADD, INC), outras são típicas de controle de execução (CJNE, JB) e, por fim, temos o exemplo de uma operação lógica.

Note também que essas operações são simples: já vimos como projetar circuitos combinacionais para fazer esse tipo de operação!

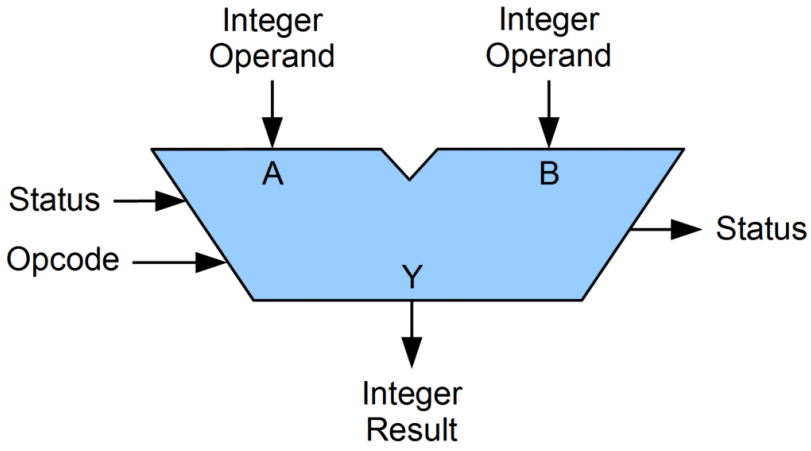
O modelo de Von Neumann.



A **memória** pode guardar tanto o programa a ser executado quanto os dados (resultados).

A **unidade de controle** coordena os operandos e as operações, buscando os dados na memória e disponibilizando para que outros circuitos executem essas operações.

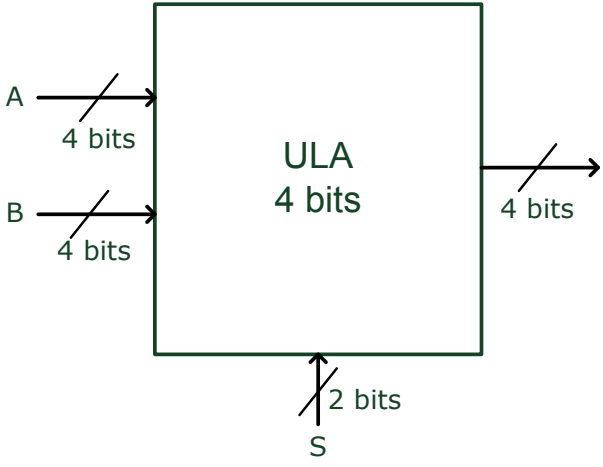
A **unidade lógico-aritmética** executa propriamente essas operações, retornando um valor.


Integer
Operand
Integer
Operand
A
B
Status
Opcode
Status
Y
Integer
Result

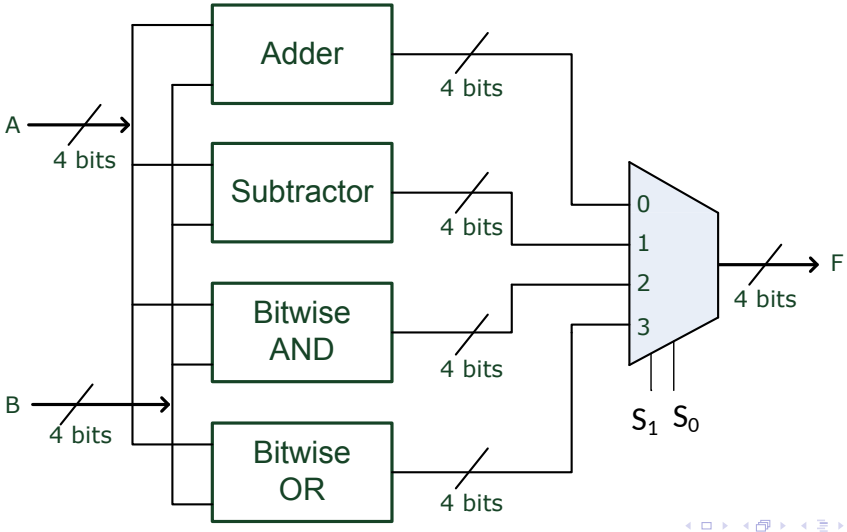
Uma unidade lógico-aritmética (ULA) (ou ALU - *arithmetic logic unit*) é um circuito combinacional que pode realizar uma série de operações aritméticas e/ou lógicas em um par de operandos de n-bits. A operação é especificada em suas entradas de seleção.

Considere uma ULA com dois operandos de 4 bits com dois bits de seleção que execute as seguintes operações com seus operandos:

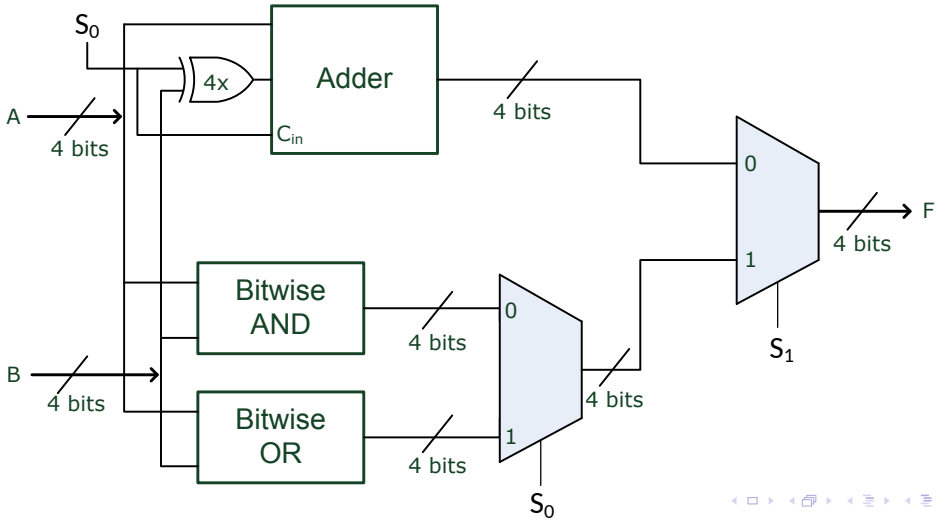
Entradas		
$\mathbf{S_1}$	$\mathbf{S_0}$	Função
0	**0**	$F = A + B$
0	**1**	$F = A - B$
1	**0**	A **and** B
1	**1**	A **or** B



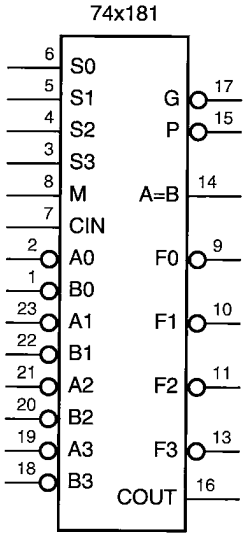
Individualmente sabemos fazer todos estes circuitos. Portanto, uma primeira solução poderia ser:



Já vimos que essa primeira solução não é ideal, pois utiliza mais somadores do que o necessário. Que tal usar a representação de números com sinal em complemento de 2?



Standard MSI ALUs: 74x181



O CI 74x181 é uma ALU de 4 bits. Este CI tem:

- Dois operandos de 4 bits A e B (ativos em nível baixo).
- Um operando de 1 bit ativo em nível alto C_{in}.
- Um seletor entre operações lógicas e aritméticas M.
- Quatro bits de seleção para a operação a ser realizada S.
- Quatro bits de saída F.
- Um bit de *carry out* C_{out} e duas saídas de grupo de *carry lookahead*.
- Um bit de igualdade $A = B$.

Standard MSI ALUs: 74x181

Entradas				Função	
S_3	S_2	S_1	S_0	$M = 0$ (arithmetic)	$M = 1$ (logic)
0	**0**	**0**	**0**	F = A minus 1 plus CIN	F = A'
0	**0**	**0**	**1**	F = A · B minus 1 plus CIN	F = A' + B'
0	**0**	**1**	**0**	F = A · B' minus 1 plus CIN	F = A' + B
0	**0**	**1**	**1**	F = 1111 plus CIN	F = 1111
0	**1**	**0**	**0**	F = A plus (A + B') plus CIN	F = A' · B'
0	**1**	**0**	**1**	F = A · B plus (A+B') plus CIN	F = B'
0	**1**	**1**	**0**	F = A minus B minus 1 plus CIN	F = A ⊕ B'
0	**1**	**1**	**1**	F = A + B' plus CIN	F = A + B'
1	**0**	**0**	**0**	F = A plus (A + B) plus CIN	F = A' · B
1	**0**	**0**	**1**	F = A plus B plus CIN	F = A ⊕ B
1	**0**	**1**	**0**	F = A · B' plus (A+B) plus CIN	F = B
1	**0**	**1**	**1**	F = A + B plus CIN	F = A + B
1	**1**	**0**	**0**	F = A plus A plus CIN	F = 0000
1	**1**	**0**	**1**	F = A · B plus A plus CIN	F = A · B'
1	**1**	**1**	**0**	F = A · B' plus A plus CIN	F = A · B
1	**1**	**1**	**1**	F = A plus CIN	F = A

Onde: **plus** é soma aritmética, **minus** é subtração aritmética, e os símbolos ·, + e ⊕ se referem aos símbolos lógicos operados bit a bit.

Para somar A e B, usamos $M = 0$ e $S = 1001$ (usando a função A plus B plus CIN).

Para fazer $A - B$ em complemento 2, usamos $M = 0$, $S = 0110$ e $CIN = 1$ (usando a função A minus B minus 1 plus CIN).

Para decrementar A, fazemos $M = 0$, $S = 0000$ e $CIN = 0$ (usando a função A minus 1 plus CIN).

Algumas operações podem não fazer muito sentido, mas "vem de graça", como por exemplo "F = A · B' plus (A+B) plus CIN".

Entradas				Função	
S_3	S_2	S_1	S_0	$M = 0$ (arithmetic)	$M = 1$ (logic)
0	**0**	**0**	**0**	F = A minus 1 plus CIN	F = A'
0	**0**	**0**	**1**	F = A · B minus 1 plus CIN	F = A' + B'
0	**0**	**1**	**0**	F = A · B' minus 1 plus CIN	F = A' + B
0	**0**	**1**	**1**	F = 1111 plus CIN	F = 1111
0	**1**	**0**	**0**	F = A plus (A + B') plus CIN	F = A' · B'
0	**1**	**0**	**1**	F = A · B plus (A+B') plus CIN	F = B'
0	**1**	**1**	**0**	F = A minus B minus 1 plus CIN	F = A ⊕ B'
0	**1**	**1**	**1**	F = A + B' plus CIN	F = A + B'
1	**0**	**0**	**0**	F = A plus (A + B) plus CIN	F = A' · B
1	**0**	**0**	**1**	F = A plus B plus CIN	F = A ⊕ B
1	**0**	**1**	**0**	F = A · B' plus (A+B) plus CIN	F = B
1	**0**	**1**	**1**	F = A + B plus CIN	F = A + B
1	**1**	**0**	**0**	F = A plus A plus CIN	F = 0000
1	**1**	**0**	**1**	F = A · B plus A plus CIN	F = A · B'
1	**1**	**1**	**0**	F = A · B' plus A plus CIN	F = A · B
1	**1**	**1**	**1**	F = A plus CIN	F = A

Outras ALU de 8 bits mais simples são os CIs 74x381 e 74x382. A única diferença entre esses CIs é que o 74x381 tem saídas para *group carry lookahead*.

Entradas			
S_2	S_1	S_0	Função
0	**0**	**0**	F = 0000
0	**0**	**1**	F = B minus A minus 1 plus CIN
0	**1**	**0**	F = A minus B minus 1 plus CIN
0	**1**	**1**	F = A plus B plus CIN
1	**0**	**0**	F = A $\oplus$ B
1	**0**	**1**	F = A $+$ B
1	**1**	**0**	F = A $\cdot$ B
1	**1**	**1**	F = 1111

Onde: **plus** é soma aritmética, **minus** é subtração aritmética, e os símbolos $\cdot$, $+$ e $\oplus$ se referem aos símbolos lógicos operados bit a bit.

Standard MSI Adders: 74x381 e 74x382