B.5 Construindo uma unidade lógica e aritmética

*ALU n. [**A**rthritic **L**ogic **U**nit ou (raro) **A**rithmetic **L**ogic **U**nit] Um gerador de números aleatórios fornecido por padrão com todos os sistemas computacionais.*
Stan Kelly-Bootle, *The Devil's DP Dictionary*, 1981

A *unidade lógica e aritmética* (*ALU – Arithmetic Logic Unit*) é o músculo do computador, o dispositivo que realiza as operações aritméticas, como adição e subtração, ou as operações lógicas, como AND e OR. Esta seção constrói uma ALU a partir de quatro blocos de montagem do hardware (portas AND e OR, inversores e multiplexadores) e ilustra como funciona a lógica combinacional. Na próxima seção, veremos como a adição pode ser agilizada por meio de projetos mais inteligentes.

Como a word do MIP tem 32 bits de largura, precisamos de uma ALU de 32 bits. Vamos supor que iremos conectar 32 ALUs de 1 bit para criar a ALU desejada. Portanto, vamos começar construindo uma ALU de 1 bit.

Uma ALU de 1 bit

As operações lógicas são as mais fáceis, pois são mapeadas diretamente nos componentes de hardware da Figura B.2.1.

A unidade lógica de 1 bit para AND e OR se parece com a Figura B.5.1. O multiplexador à direita, então, seleciona *a* AND *b* ou *a* OR *b*, dependendo se o valor de *Operação* é 0 ou 1. A linha que controla o multiplexador aparece em destaque para distingui-la das linhas com dados. Observe que renomeamos as linhas de controle e a saída do multiplexador para lhes dar nomes que refletem a função da ALU.

A próxima função a incluir é a adição. Um somador precisa ter duas entradas para os operandos e uma saída de único bit para a soma. É preciso haver uma segunda saída para o carry, chamada *CarryOut*. Como o CarryOut do somador vizinho precisa ser incluído como uma entrada, precisamos de uma terceira entrada. Essa entrada é chamada *CarryIn*. A Figura B.5.2 mostra as entradas e as saídas de um somador de 1 bit. Como sabemos o que a adição precisa fazer, podemos especificar as saídas dessa "caixa preta" com base em suas entradas, como a Figura B.5.3 demonstra.

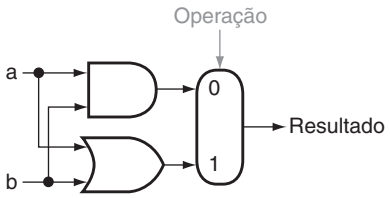

FIGURA B.5.1 A unidade lógica de 1 bit para AND e OR.

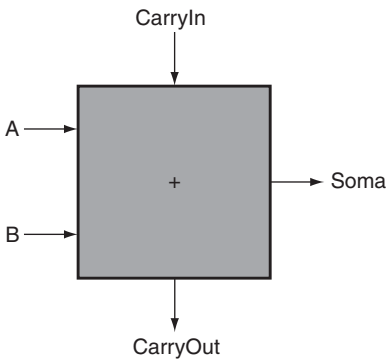

FIGURA B.5.2 Um somador de 1 bit. Esse somador é chamado de somador completo; ele também é chamado somador (3,2), pois tem 3 entradas e 2 saídas. Um somador com apenas as entradas a e b é chamado somador (2,2) ou meio somador.

Entradas			Saídas		
a	B	CarryIn	CarryOut	Soma	Comentários
0	0	0	0	0	$0 + 0 + 0 = 00_{bin}$
0	0	1	0	1	$0 + 0 + 1 = 01_{bin}$
0	1	0	0	1	$0 + 1 + 0 = 01_{bin}$
0	1	1	1	0	$0 + 1 + 1 = 10_{bin}$
1	0	0	0	1	$1 + 0 + 0 = 01_{bin}$
1	0	1	1	0	$1 + 0 + 1 = 10_{bin}$
1	1	0	1	0	$1 + 1 + 0 = 10_{bin}$
1	1	1	1	1	$1 + 1 + 1 = 11_{bin}$

FIGURA B.5.3 Especificação de entrada e saída para um somador de 1 bit.

Podemos expressar as funções de saída CarryOut e Soma como equações lógicas, e essas equações, por sua vez, podem ser implementadas com portas lógicas. Vamos realizar um CarryOut. A Figura B.5.4 mostra os valores das entradas quando CarryOut é 1.

Podemos transformar essa tabela verdade em uma equação lógica.

$$\text{CarryOut} = (b \,.\, \text{CarryIn}) + (a \,.\, \text{CarryIn}) + (a \,.\, b) + (a \,.\, b \,.\, \text{CarryIn})$$

Entradas		
a	b	CarryIn
0	1	1
1	0	1
1	1	0
1	1	1

FIGURA B.5.4 Valores das entradas quando CarryOut é 1.

Se a . b. CarryIn for verdadeiro, então todos os outros três termos também precisam ser verdadeiros, de modo que podemos omitir esse último termo correspondente à quarta linha da tabela. Assim, podemos simplificar a equação para

$$\text{CarryOut} = (b \,.\, \text{CarryIn}) + (a \,.\, \text{CarryIn}) + (a \,.\, b)$$

A Figura B.5.5 mostra que o hardware dentro da caixa preta do somador para CarryOut consiste em três portas AND e uma porta OR. As três portas AND correspondem exatamente aos três termos entre parênteses da fórmula anterior para CarryOut, e a porta OR soma os três termos.

O bit Soma é ligado quando exatamente uma entrada é 1 ou quando todas as três entradas são 1. A Soma resulta em uma equação Booleana complexa (lembre-se de que $\bar{a}$ significa NOT a):

$$\text{Soma} = (a \,.\, \bar{b} \,.\, \overline{\text{CarryIn}}) + (\bar{a} \,.\, b \,.\, \overline{\text{CarryIn}}) + (\bar{a} \,.\, \bar{b} \,.\, \text{CarryIn}) + (a \,.\, b \,.\, \text{CarryIn})$$

O desenho da lógica para o bit Soma na caixa preta do somador fica como um exercício.

A Figura B.5.6 mostra uma ALU e 1 bit derivada da combinação do somador com os componentes anteriores. Às vezes, os projetistas também querem que a ALU realize mais algumas operações simples, como gerar 0. O modo mais fácil de somar uma operação é expandir o multiplexador controlado pela linha Operação e, para este exemplo, conectar 0 diretamente à nova entrada desse multiplexador expandido.

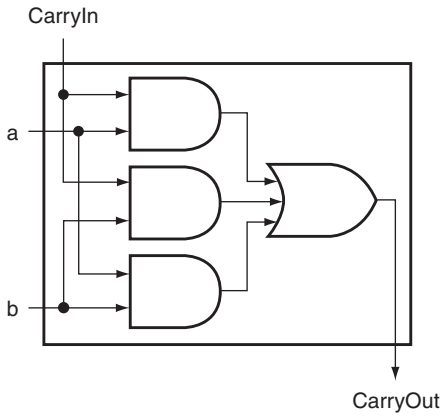

FIGURA B.5.5 Hardware do somador para o sinal CarryOut. O restante do hardware do somador é a lógica para a saída de Soma dada na equação da página B-30.

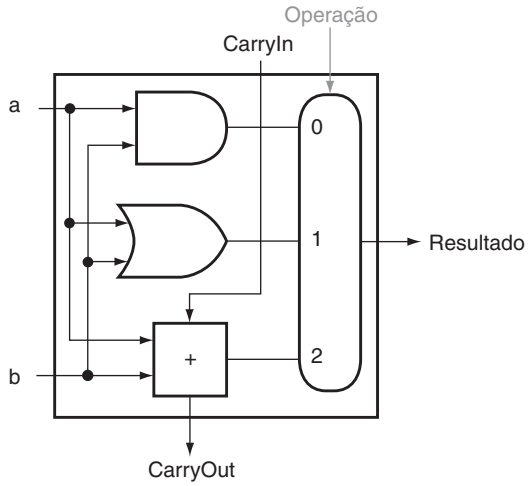

FIGURA B.5.6 Uma ALU de 1 bit realiza AND, OR e adição (ver Figura B.5.5).

Uma ALU de 32 bits

Agora que completamos a ALU de 1 bit, a ALU completa de 32 bits é criada conectando "caixas pretas" adjacentes. Usando *xi* para indicar o *i-ésimo* bit de *x*, a Figura B.5.7 mostra uma ALU de 32 bits. Assim como uma única pedra pode causar ondulações partido da costa de um lago tranqüilo, um único carry do bit menos significativo (Result0) pode causar ondulações por todo o somador, levando a uma carry do bit mais significativo (Result31). Logo, o somador criado ligando diretamente os carries de somadores de 1 bit é chamado de somador de *carry por ondulação*. Veremos um modo rápido de conectar os somadores de 1 bit a partir da página B-30.

A subtração é o mesmo que a adição da versão negativa de um operando, e é assim que os somadores realizam a subtração. Lembre-se de que o atalho para negar um número em complemento a dois é inverter cada bit (às vezes chamado de *complemento a um*) e depois somar 1. Para inverter cada bit, simplesmente acrescentamos um multiplexador 2:1 que escolhe entre b e $\overline{b}$, como mostra a Figura B.5.8.

Suponha que conectemos 32 dessas ALUs de 1 bit, como fizemos na Figura B.5.7. O multiplexador adicionado dá a opção de b ou seu valor invertido, dependendo de Binvert, mas essa é apenas uma etapa na negação de um número em complemento a dois. Observe que o bit menos significativo ainda possui um sinal CarryIn, embora seja desnecessário para a adição. O que acontece se definir-

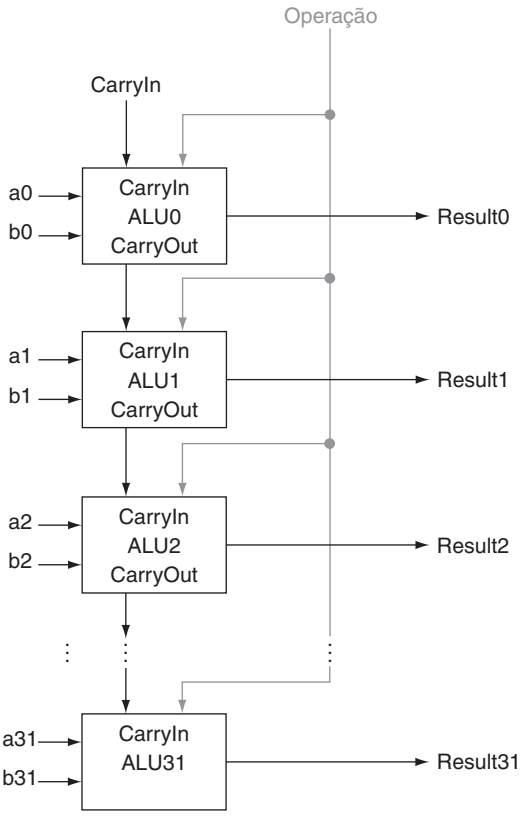

FIGURA B.5.7 Uma ALU de 32 bits construída a partir de 32 ALUs de 1 bit. O CarryOut de um bit é conectado ao CarryIn do próximo bit mais significativo. Essa organização é chamada de carry por ondulação.

mos esse CarryIn como 1 em vez de 0? O somador, então, calculará a + b + 1. Selecionando a versão invertida de b, obtemos exatamente o que queremos:

$$a + \overline{b} + 1 = a + (\overline{b} + 1) = a + (-b) = a - b$$

A simplicidade do projeto de hardware de um somador em complemento a dois ajuda a explicar por que a representação em complemento a dois tornou-se um padrão universal para a aritmética computacional com inteiros.

Uma ALU no MIPS também precisa de uma função NOR. Em vez de acrescentar uma porta separada para NOR, podemos reutilizar grande parte do hardware já existente na ALU, como fizemos para a subtração. A idéia vem da seguinte tabela verdade sobre NOR:

$$\overline{(a + b)} = \overline{a} \,.\, \overline{b}$$

Ou seja, NOT (a OR b) é equivalente a NOT a AND NOT b. Esse fato é chamado de teorema de DeMorgan e é explorado nos exercícios com mais profundidade.

Como temos AND e NOT b, só precisamos acrescentar NOT a à ALU. A Figura B.5.9 mostra essa mudança.

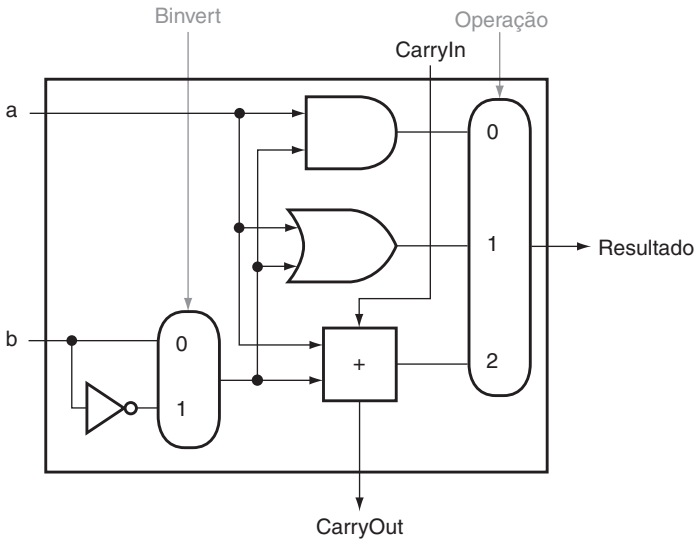

FIGURA B.5.8 Uma ALU de 1 bit que realiza AND, OR e adição entre a e b ou a e $\overline{b}$. Selecionando $\overline{b}$ (Binvert = 1) e definindo CarryIn como 1 no bit menos significativo da ALU, obtemos a subtração em complemento a dois de b a partir de a, em vez da adição de b e a.

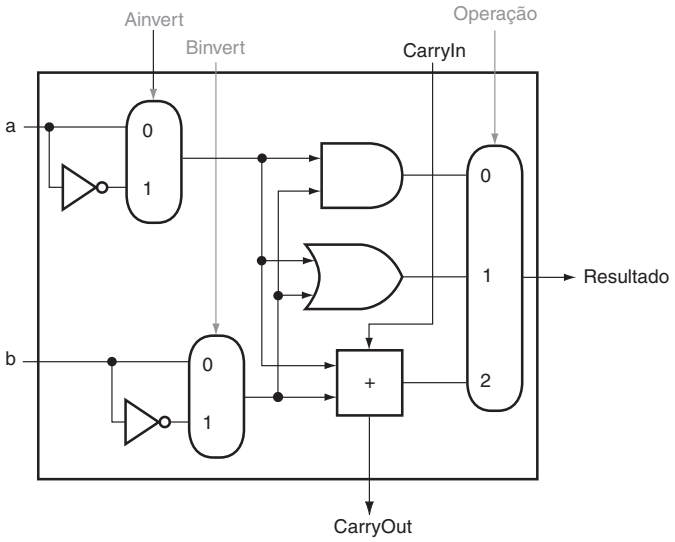

FIGURA B.5.9 Uma ALU de 1 bit que realiza AND, OR e adição entre a e b ou $\overline{a}$ e $\overline{b}$. Selecionando $\overline{a}$ (Ainvert = 1) e $\overline{b}$ (Binvert = 1), obtemos a NOR b, em vez de a AND b.

Ajustando a ALU de 32 bits ao MIPS

Essas quatro operações – adição, subtração, AND, OR – são encontradas na ALU de quase todo computador, e as operações da maioria das instruções MIPS podem ser realizadas por essa ALU. Mas o projeto da ALU está incompleto.

Uma instrução que ainda precisa de suporte é a instrução "set on less than" (`slt`). Lembre-se de que a operação produz 1 se rs < rt, ou 0 em caso contrário. Conseqüentemente, `slt` colocará todos os bits, menos o bit menos significativo, em 0, com o valor do bit menos significativo definido de acordo com a comparação. Para a ALU realizar `slt`, primeiro precisamos expandir o multiplexador de três entradas da Figura B.5.8 para acrescentar uma entrada para o resultado de `slt`. Chamamos essa nova entrada de *Less* e a usamos apenas para `slt`.

O desenho superior da Figura B.5.10 mostra a nova ALU de 1 bit com o multiplexador expandido. A partir da descrição de `slt` anterior, temos de conectar 0 à entrada Less para os 31 bits superiores da

ALU, pois esses bits sempre serão 0. O que falta considerar é como comparar e definir o valor do *bit menos significativo* para instruções "set on less than".

O que acontece se subtrairmos b de a? Se a diferença for negativa, então a < b, pois

$$(a - b) < 0 \Rightarrow ((a - b) + b) < (0 + b)$$
$$\Rightarrow a < b$$

Queremos que o bit menos significativo de uma operação "set on less than" seja 1 se a < b; ou seja, 1 se a – b for negativo e 0 se for positivo. Esse resultado desejado corresponde exatamente aos valores do bit de sinal: 1 significa negativo e 0 significa positivo. Seguindo essa linha de argumento, só precisamos conectar o bit de sinal da saída do somador ao bit menos significativo para obter "set on less than".

Infelizmente, a saída Result do bit da ALU mais significativo no topo da Figura B.5.10 para a operação `slt` *não* é a saída do somador; a saída da ALU para a operação `slt` é obviamente o valor de entrada Less.

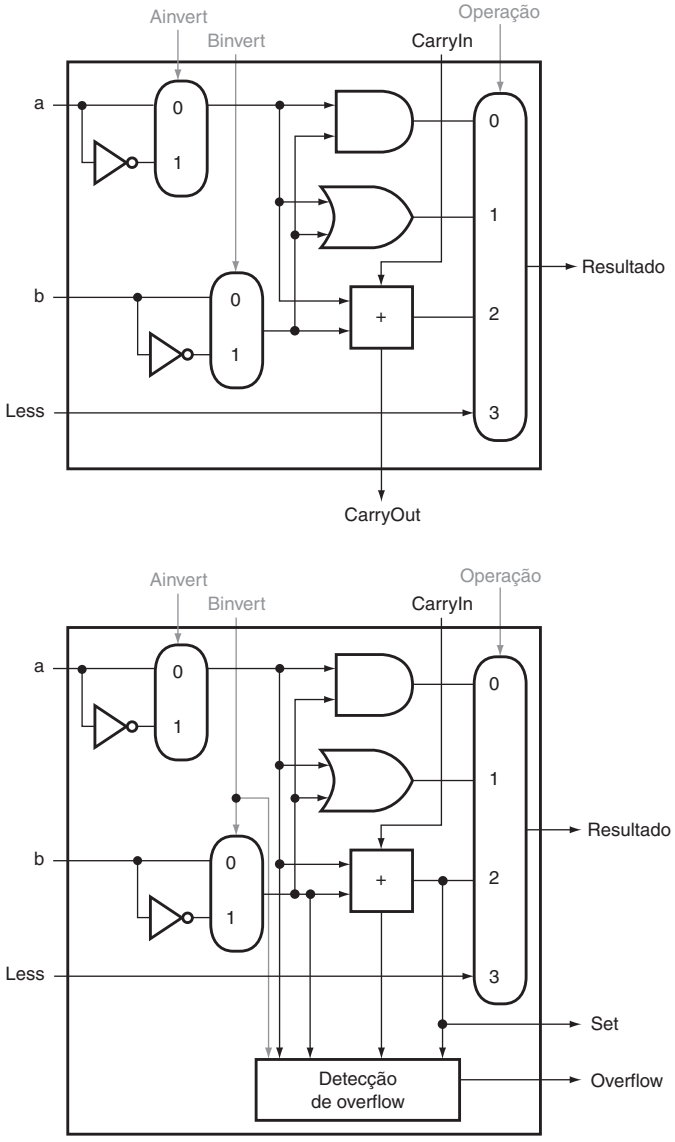

FIGURA B.5.10 (Superior) Uma ALU de 1 bit que realiza AND, OR e adição entre a e b ou $\overline{b}$, e (inferior) uma ALU de 1 bit para o bit mais significativo. O desenho superior inclui uma entrada direta que está conectada para realizar a operação "set on less than" (ver Figura B.5.11); o desenho inferior possui uma saída direta do somador para a comparação "less than", chamada Set. (Veja, no Exercício 3.24, como calcular o overflow com menos entradas.)

Assim, precisamos de uma nova ALU de 1 bit, para o bit mais significativo, a qual possui um bit de saída extra: a saída do somador. O desenho inferior da Figura B.5.10 mostra o projeto, com essa nova linha de saída do somador chamada *Set*, e usada apenas para `slt`. Como precisamos de uma ALU especial para o bit mais significativo, acrescentamos a lógica de detecção de overflow, pois também está associada a esse bit.

Infelizmente, o teste de "less than" é um pouco mais complicado do que acabamos de descrever, devido ao overflow, conforme exploramos nos exercícios. A Figura B.5.11 mostra a ALU de 32 bits.

Observe que toda vez que quisermos que a ALU subtraia, colocamos CarryIn e Binvert em 1. Para adições ou operações lógicas, queremos que as duas linhas de controle sejam 0. Portanto, podemos simplificar o controle da ALU combinando CarryIn e Binvert a uma única linha de controle, chamada *Bnegate*.

Para ajustar ainda mais a ALU ao conjunto de instruções do MIPS, temos de dar suporte a instruções de desvio condicional. Essas instruções desviam se dois registradores forem iguais ou se forem diferentes. O modo mais fácil de testar a igualdade com a ALU é subtrair b de a e depois testar se o resultado é zero, pois

$$(A - B = 0) \Rightarrow a = b$$

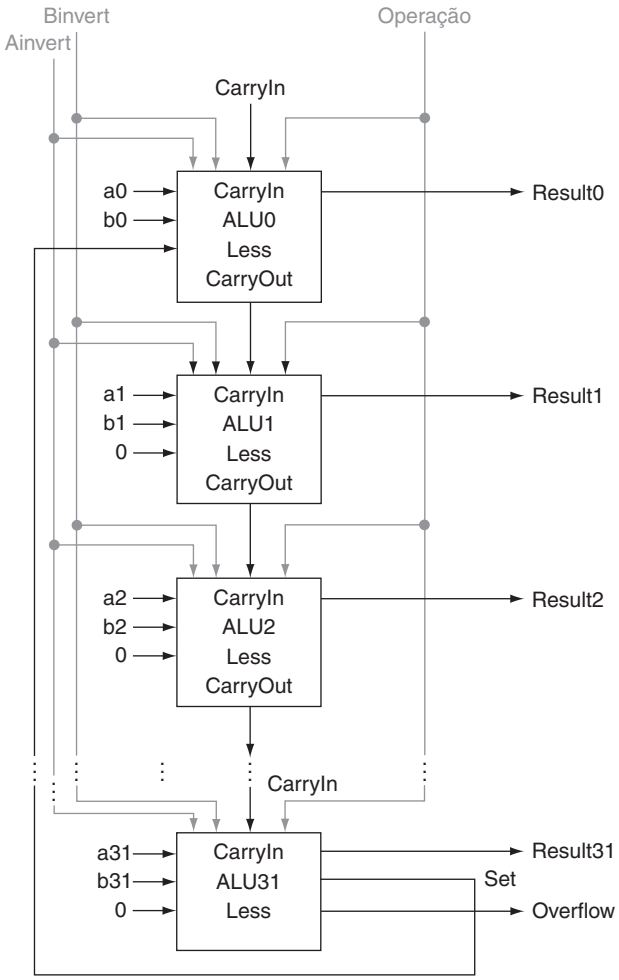

FIGURA B.5.11 Uma ALU de 32 bits construída a partir de 31 cópias da ALU de 1 bit na parte superior da Figura B.5.10 e uma ALU de 1 bit na parte inferior dessa figura. As entradas Less são conectadas a 0, exceto para o bit menos significativo, que está conectado à saída Set do bit mais significativo. Se a ALU realizar a – b e selecionarmos a entrada 3 no multiplexador da Figura B.5.10, então Result = 0 ... 001 se a < b, e Result = 0 ... 000 caso contrário.

Assim, se acrescentarmos hardware para testar se o resultado é 0, podemos testar a igualdade. O modo mais simples é realizar um OR de todas as saídas juntas e depois enviar esse sinal por um inversor:

$$\text{Zero} = \overline{(\text{Result31} + \text{Result30} + \ldots + \text{Result2} + \text{Result1} + \text{Result0})}$$

A Figura B.5.12 mostra a ALU de 32 bits revisada. Podemos pensar na combinação da linha Ainvert de 1 bit, a linha Binvert de 1 bit, e as linhas de Operação de 2 bits como linhas de controle de 4 bits para a ALU, pedindo que realize soma, subtração, AND, OR ou "set on less than". A Figura B.5.13 mostra as linhas de controle da ALU e a operação ALU correspondente.

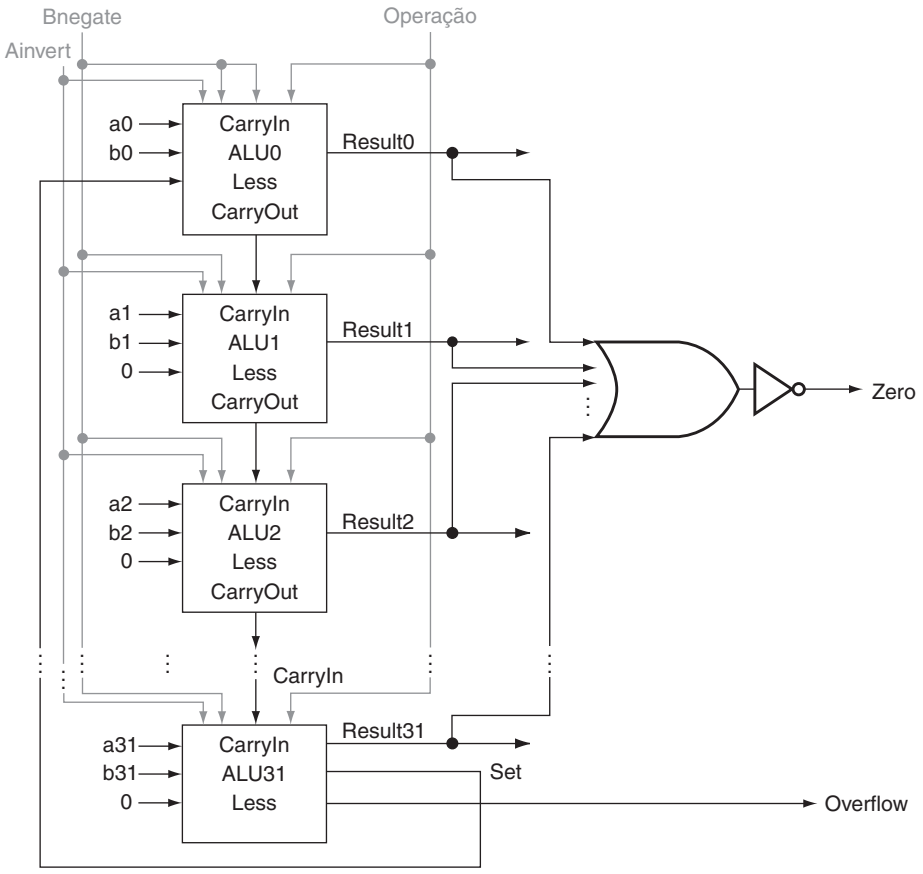

FIGURA B.5.12 A ALU final de 32 bits. Isso acrescenta um detetor de zero à Figura B.5.11.

Linhas de controle da ALU	Função
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

FIGURA B.5.13 Os valores das três linhas de controle ALU Bnegate e Operação e as operações ALU correspondentes.

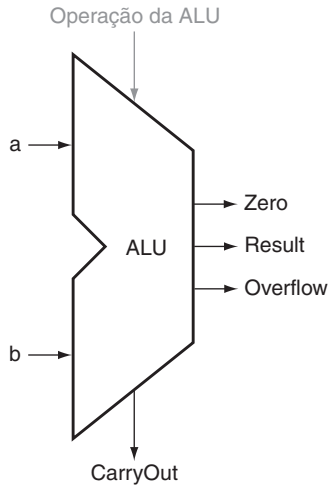

FIGURA B.5.14 O símbolo normalmente usado para representar uma ALU, como mostra a Figura B.5.12. Esse símbolo também é usado para representar um somador, de modo que normalmente é rotulado com ALU ou Adder (somador).

Finalmente, agora que vimos o que há dentro de uma ALU de 32 bits, usaremos o símbolo universal para uma ALU completa, como mostra a Figura B.5.14.

Definindo a ALU MIPS em Verilog

A Figura B.5.15 mostra como uma ALU combinacional do MIPS poderia ser especificada em Verilog; essa especificação provavelmente seria compilada com uma biblioteca de partes padrão, que oferecesse um somador, que poderia ser instanciado. Para completar, mostramos o controle da ALU para o MIPS na Figura B.5.16, que usaremos mais adiante quando montarmos uma versão Verilog do caminho de dados do MIPS no Capítulo 5.

```
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [31:0] A,B;
   output reg [31:0] ALUOut;
   output Zero;

   assign Zero = (ALUOut==0); // Zero é true se ALUOut é 0
   always @(ALUctl, A, B) begin // reavalia se isso mudar
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A – B;
         7: ALUOut <= A < B ? 1 : 0;
         12: ALUOut <= ~(A | B); // resultado é nor
         default: ALUOut <= 0;
      endcase
   end
endmodule
```

FIGURA B.5.15 Definição comportamental em Verilog de uma ALU MIPS.

```
module ALUControl (ALUOp, FuncCode, ALUCtl);

   input [1:0] ALUOp;
   input [5:0] FuncCode;
   output [3:0] reg ALUCtl;

   always case (FuncCode)

   32: ALUOp<=2;  // soma
   34: ALUOp<=6;  // subtrai
   36: ALUOP<=0;  // and
   37: ALUOp<=1;  // or
   39: ALUOp<=12; // nor
   42: ALUOp<=7;  // slt
   default: ALUOp<=15; // não deverá acontecer
   endcase
endmodule
```

FIGURA B.5.16 O controle ALU do MIPS: um peça simples da lógica de controle combinacional.

A próxima pergunta é: com que rapidez essa ALU pode somar dois operandos de 32 bits? Podemos determinar as entradas a e b, mas a entrada CarryIn depende da operação no somador de 1 bit adjacente. Se traçarmos todo o caminho pela cadeia de dependências, conectamos o bit mais significativo ao bit menos significativo, de modo que o bit mais significativo da soma precisa esperar pela avaliação *seqüencial* de todos os 32 somadores de 1 bit. Essa reação em cadeia seqüencial é muito lenta para ser usada no hardware de tempo crítico. A próxima seção explora como agilizar a adição. Esse assunto não é fundamental para a compreensão do restante do apêndice e pode ser pulado.

Verifique você mesmo Suponha que você queira acrescentar a operação NOT (a AND b), chamada NAND. Como a ALU poderia mudar para dar suporte a ela?

1. Nenhuma mudança. Você pode calcular NAND rapidamente usando a ALU atual, pois $(\overline{a \,.\, b}) = (\overline{a} + \overline{b})$ e já temos NOT a, NOT b, e OR.
2. Você precisa expandir o multiplexador grande para acrescentar outra entrada e depois acrescentar nova lógica para calcular NAND.

B.6 Adição mais rápida: Carry Lookahead

A chave para agilizar a adição é determinar o "carry in" para os bits mais significativos mais cedo. Existem diversos esquemas para antecipar o carry, de modo que o cenário de pior caso é uma função do $\log_2$ do número de bits do somador. Esses sinais antecipatórios são mais rápidos porque passam por menos portas em seqüência, mas são necessárias muito mais portas para antecipar o carry apropriado.

Uma chave para entender os esquemas de carry rápido é lembrar que, diferente do software, o hardware executa em paralelo sempre que as entradas mudam.