A Representação Numérica

Usamos de forma rotineira o sistema numérico decimal e muitas vezes nos esquecemos das regras para sua formação. Aliás, estas regras nos são tão familiares que até parecem fazer parte da natureza. O sistema de numeração calcado na base dez é natural ao ser humano pela associação com seus dez de dedos. Entretanto, levou muito tempo para o homem o desenvolvesse e é um legado das civilizações arábica e indiana. É interessante comentarmos um pouco mais sobre sistemas de numeração, para vermos que outros sistemas de numeração já foram empregados pelo homem.

Diz a história que, por volta de 3.300 AC, os Sumerianos formaram na região da Mesopotâmia a primeira civilização humana. Eles empregavam dois sistemas de numeração: base 5 e base 12. Para a base 5, usavam os dedos de uma mão para contar e os da outra para acumular a quantidade de "cinco" já contada. A contagem na base 12 era um pouco mais complexa. Supõem os historiadores que contagem se fazia com base nas falanges dos quatro dedos (3 x 4 = 12), sendo que o polegar, correndo sobre as falanges, era usado como marcador da contagem. Os dedos da outra mão eram usados como auxiliar, para marcar na base 5 a quantidade de "doze" já contada.

Da associação com os dois sistemas de contagem surge o sistema sexagesimal, ou seja, contagem base 60, resultado dos cinco dedos de uma mão vezes os doze artelhos da outra. Era o maior número que se representava com as duas mãos. Desses povos, excelentes astrônomos, herdamos sistemas de numeração que empregamos para marcar o tempo e graus em horas, minutos e segundos.

Quando se trabalha com programação precisa-se de uma forma simples de representar *bits* e naturalmente surge a idéia de empregar a base 2, com os símbolos "0" e "1". Porém, a escrita com tais símbolos é longa e monótona. Por isso, buscamse por outras bases que forneçam uma escrita compacta, mas mantendo uma forte associação com a base 2. Os melhores candidatos são a base 8 (octal), e a base hexadecimal (16) que será estudada a seguir. O importante é que fique claro que se usam essas bases como uma forma prática para representar números binários.

A.1. Representação Binária e Hexadecimal

No cotidiano usamos 10 símbolos (0, 1, 2, 3,4, 5, 6, 7, 8 e 9) arrumados em uma notação posicional. Cada um deles indica o coeficiente a ser multiplicado pela potência de 10 que é marcada pela posição. A Figura A.1 apresenta a notação posicional para o número 2017.

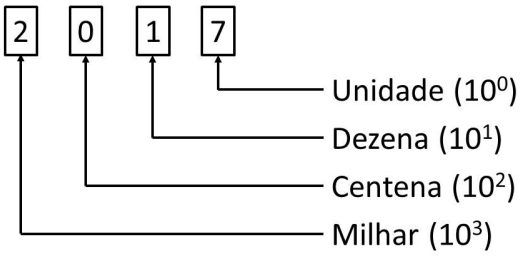

Figura A.1. Notação posicional na base 10, aplicada ao número 2017.

Para esclarecer ainda mais este conceito, o número 2017 é obtido pela expressão a seguir:

$$2017 = 2 \times 10^3 + 0 \times 10^2 + 1 \times 10^1 + 7 \times 10^0.$$

Os números são expressos usando potências de dez multiplicadas pelos coeficientes apropriados e, por isso, nosso sistema é denominado de **raiz dez** ou **base dez**. É possível, sendo muitas vezes útil, usar outras bases, como a **base dois**. Isto é vantajoso, pois, com o sistema binário, pode-se arranjar uma correspondência direta entre os números binários e as variáveis binárias (booleanas). Para evitar confusões, convenciona-se que todo número é decimal, a não ser que se diga o contrário. Aqui neste texto, os números binários são marcados pela letra **b** ou **B** colocada no seu final. É claro que, quando a base for extremamente óbvia, pode-se omitir a letra **b** para beneficiar a clareza do texto. É importante deixar claro que a linguagem C, e também o ambiente de programação do Arduino, não preveem representação em base 2. A expressão a seguir apresenta o número 37 representado em binário.

$$37 = 100101b = 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0.$$

Agora, a pergunta óbvia é: como converter números decimais para a representação binária? A resposta é bem simples e está apresentada na Figura A.2, onde se divide sucessivamente o número a converter pela base, enquanto os quocientes intermediários forem maiores que ela, e depois tomam-se os restos intermediários e o último quociente. Julgando pela explicação, parece um procedimento complicado, mas a análise da figura demonstra o contrário.

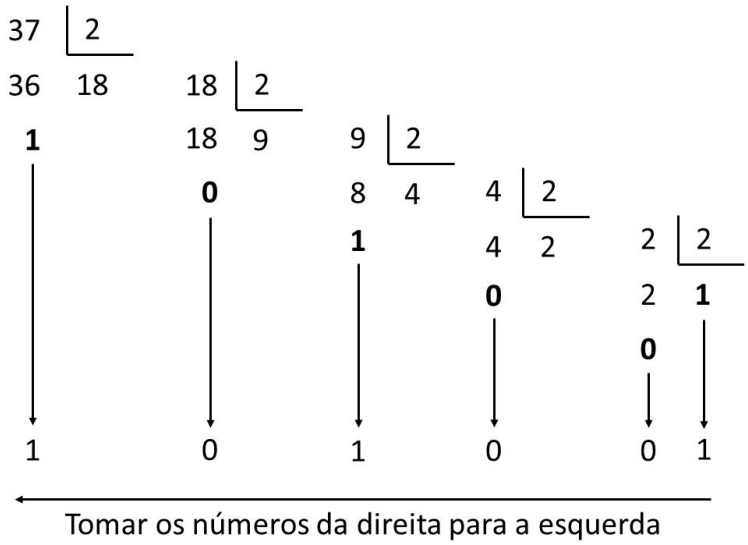

Figura A.2. Conversão da base dez para a base dois: 37d = 100101b

A Figura A.3 apresenta os primeiros 32 números binários, junto com seus valores na base 10. De posse desta tabela, o leitor deve praticar alguns exercícios de conversão de números da base 10 para a base 2.

Tabela A.1: Os primeiros 32 números binários

Base 10	Base 2	Base 10	Base 2	Base 10	Base 2	Base 10	Base 2
0	00000b	8	01000b	16	10000b	24	10000b
1	00001b	9	01001b	17	10001b	25	10001b
2	00010b	10	01010b	18	10010b	26	10010b
3	00011b	11	01011b	19	10011b	27	10011b
4	00100b	12	01100b	20	10100b	28	10100b
5	00101b	13	01101b	21	10101b	29	10101b
6	00110b	14	01110b	22	10110b	30	10110b
7	00111b	15	01111b	23	10111b	31	10111b

Uma particularidade da base 2, que já deve ter sido notada, é a quantidade de dígitos, que muitas vezes dificulta a leitura e facilita a ocorrência de erros. Por exemplo: para o número 19, foram necessários 5 dígitos binários. E para um número maior? Como é a representação do número 2017 na base 2? A resposta é: 2017d = 11111100001A.

Realmente, fica uma enorme sequência de uns e zeros, dificultando a escrita e a leitura. Um convite ao erro! Existe outra base que permite notação compacta e que ainda possui

uma forte relação para com a base binária. Esta base é a hexadecimal. Como o número da base (16) é potência de dois, ela tem uma forte ligação (como será verificado) com o sistema binário. Com a base hexadecimal, usam-se 16 símbolos diferentes. Pode-se usar os dez símbolos do sistema de numeração decimal (0, 1, 2, ..., 9), adicionando 6 símbolos extras, que, por simplicidade, são as 6 primeiras letras do alfabeto, conforme ilustrado na Tabela A.2. Os números hexadecimais serão caracterizados pelo prefixo "0x", que é usado em C. Assim, o número 2017 na base 16 é: 2016d = 111 1110 0001 b = 0x7E1.

É fácil imaginar como é feita a conversão da base 10 para a base 16. Basta realizar sucessivas divisões pelo número 16. Isto está demonstrado na Figura A.3. Já a conversão de binário para hexadecimal é extremamente simples, basta agrupar, a partir da direita, os números binários em grupos de 4 dígitos, cada grupo gerando um dígito hexadecimal. Isto também pode ser observado na Figura A.3. A volta também é verdadeira, ou seja, para converter um número base 16 para base 2 bastar substituir cada dígito hexadecimal pelos seus 4 dígitos binários correspondentes.

Tabela A.2. Os 16 Símbolos para a Notação Hexadecimal

Base 10	Base 2	Base 16
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

Base 10	Base 2	Base 16
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

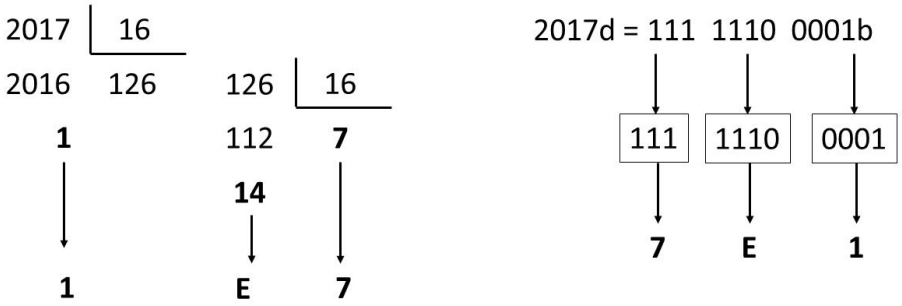

Figura A.3. Conversão da base 10 para base 16 e conversão da base 2 para a base 16.

O dígito binário, que é a menor unidade capaz de ser representada por um circuito digital, recebe o nome de *bit*, que é a contração de duas palavras inglesas *BInary digiT*. Por isso, o dígito binário é também chamado de *bit*. Estamos acostumados a agrupar os números

decimais em conjuntos de 3 dígitos. Por exemplo, ao invés do número 673286097830, é preferível escrever 673.286.097.830. A mesma coisa é feita com os números em representação binária, só que o agrupamento é de 4 em 4 para facilitar a conversão para hexadecimal. Ao agrupamento de 4 dígitos binários dá-se o nome de *nibble*. Já o processamento feito pelos computadores está baseado em agrupamentos de 8 *bits*, cada um destes recebendo o nome de *byte*. Notar que cada *byte* possui dois dígitos hexadecimais. Também é útil trabalhar com grupos de 16, 32 ou 64 *bits*. A Tabela A.3 apresenta os agrupamentos usuais e algumas de suas denominações.

Tabela A.3: Agrupamentos usados em microprocessamento

Nome	Dígitos Binários	Dígitos Hexadecimais	Nome Alternativo
bit	1	-	
nibble	4	1	
byte	8	2	
word 16	16	4	*word*
word 32	32	8	*doubleword*
word 64	64	16	*quadword*

A.2. Soma Aritmética com Números Binários

A mesma operação de soma aritmética que se faz com os números decimais é aplicada aos números binários, com os cuidados que agora existem apenas dois dígitos e que não se pode esquecer do "**vai um**" que, em inglês, tem o nome de "**carry**", já comumente consagrado na literatura técnica nacional. A Tabela A.4 apresenta os quatro casos possíveis quando se somam dois dígitos binários.

Tabela A.4: Soma de dois dígitos binários

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	0 e "vai um"

Devemos ter cuidado para não confundirmos a soma aritmética com a soma lógica (booleana). Quando se usam funções OR e AND, estão sendo realizadas operações lógicas. Porém, aqui neste tópico, estamos tratando de operações aritméticas, razão pela

qual 1 + 1 é igual a zero e aparece o "vai um". A Figura A.4 apresenta duas somas com números em representação binária. O leitor é convidado a repetir esta operação.

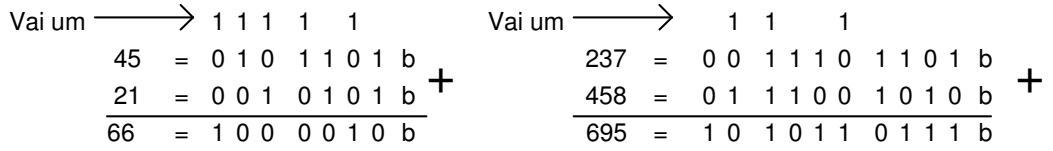

Figura A.4. Soma de números em representação binária.

A mesma soma pode ser feita com dígitos hexadecimais, tomando cuidado com as somas intermediárias. A Figura A.5 apresenta duas somas com dígitos hexadecimais.

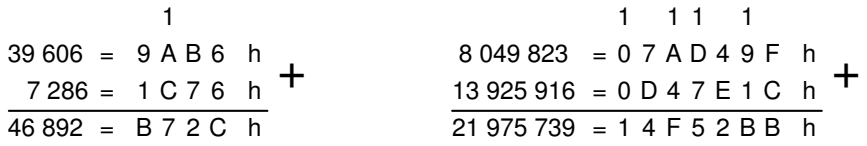

Figura A.5. Soma de números em representação hexadecimal.

A.3. Números Negativos e Subtração

O título deste tópico associa os números negativos à subtração, pois será mostrada uma representação negativa para auxiliar na operação de subtração. Da mesma forma, como existem as regras para a soma de dois números binários, também existem as regras para a subtração (isso é extremamente óbvio). Por exemplo, podemos subtrair 3 do número 5; para tanto, usamos as regras da subtração. Porém, o que se deseja neste ponto é algo um pouco diferente: encontrar uma representação para o "três negativo" (-3), tal que somada ao número 5 resulte na operação 5 + (-3) = 5 – 3 = 2.

5 - 3 = 2 → subtraindo 3 do número 5
5 + (-3) = 2 → somando -3 (três negativo) ao número 5

Por que essa preocupação? Por que não realizar logo a subtração? Nos projetos digitais, um ponto muito importante é a complexidade do circuito. Quanto maior o circuito, mais caro ele é e também maior é a probabilidade de falhas. Por isso, ao invés de se projetar dois circuitos distintos: um somador e um subtrator, projeta-se único *hardware* que, entre outras coisas, faz operações de soma e subtração. Com a representação de números negativos, o somador também poder ser usado para realizar as subtrações.

O primeiro conceito útil para o entendimento da representação de números negativos é o de **Complemento Um**. É muito simples calcular este complemento, bastando inverter todos os *bits*:

se o número 3 é representado por: 0011;
seu complemento um será: 1100;

se o número 458 é representado por: 0001 1100 1010;
seu complemento um será: 1110 0011 0101.

No dia a dia, quando se diz que uma pessoa é totalmente inoperante, diz-se que "ela é um zero à esquerda". É óbvio que esta expressão vem do fato de que, à esquerda de um número positivo, podem-se colocar tantos zeros quanto se queira, pois eles não têm a menor importância. De forma semelhante, será mostrado que, à esquerda de um número negativo, pode-se por tantos uns quantos se queira.

Os circuitos digitais sempre operam com uma quantidade limitada de *bits*, tais como os valores típicos 4, 8, 16, 32, e 64. Para evitar a monotonia dos cálculos longos, adotaremos, neste momento, o trabalho apenas com 4 *bits*.

Na verdade, os números negativos são designados usando-se uma representação em **Complemento Dois**. A definição de complemento 2 de um número N com B *bits* na parte inteira é:

$$(N)_{2,C} = 2^B - N.$$

Assim, o complemento 2 dos números inteiros de 0 a 15, usando 4 *bits* na parte inteira, são respectivamente 16 a 1. Ao adotar-se a representação de números negativos usando complemento 2, interpreta-se o *bit* mais a esquerda (chamado de *bit* mais significativo) como o *bit* de sinal. Se o *bit* de sinal for 0, o número é positivo; se for 1, é negativo. Desta forma, como achar a representação de -3? Basta calcular o complemento 2 dele, que é 13. Portanto, (-3) é representado como 1101.

Como mostrado na Figura A.6, existe uma dualidade na adoção da representação dos números. Caso se use uma representação sem sinal, os números de 4 *bits* variam de 0 a 15. Caso se use uma representação com sinal (complemento a dois), os números variam de -8 a +7. Para números inteiros de 8 *bits* sem sinal, a variação vai de 0 a 255, enquanto que, para inteiros com sinal, vai de -128 a +127.Uma curiosidade, podemos dizer que "0 é positivo", pois seu *bit* de sinal é igual a zero.

É importante ficar claro que tudo se resume em como interpretamos o número. Na representação inteira sem sinal, não há dúvida de qual é o número 5 (0101b) e de qual é o número 11 (1011b). Já na representação com sinal, o número +5 é representado por 0101b e o -5 representado por 1101b (que era o número 13 sem sinal). Note que o número 13 passou a representar o -5. Assim, não basta apresentar um número, é preciso indicar se é sem sinal ou com sinal. Fazemos isso quando declaramos variáveis:

```
char i;                 //de −128 até +127
unsigned char i;        //de 0 até 255
int i;                  //de −32.768 até +32.767
unsigned int i;         //de 0 até 65.535
```

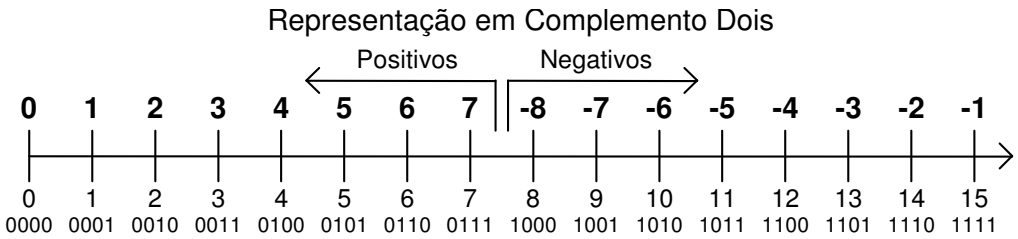

Figura A.6. A representação em Complemento Dois para inteiros de 4 bits.

Um truque interessante para calcular o complemento dois de um número é efetuar o seu complemento um e depois somar o *bit* 1 ao *bit* mais à direita (chamado de *bit* menos significativo). A Figura A.7 exemplifica o complemento dois para dois números.

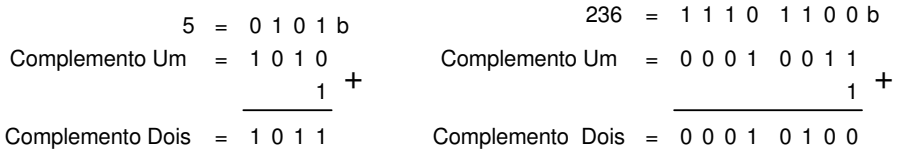

Figura A.7. Cálculo do complemento dois de alguns números.

Existe outro algoritmo muito útil para o cálculo do complemento dois: começar a reescrever o número binário da direita para a esquerda e quando encontrar o primeiro dígito 1, repetir o mesmo e complementar todos os demais dígitos. Mesmo se o dígito mais à direita for 1, repeti-lo e inverter todos os demais. A Figura A.8 ilustra este algoritmo.

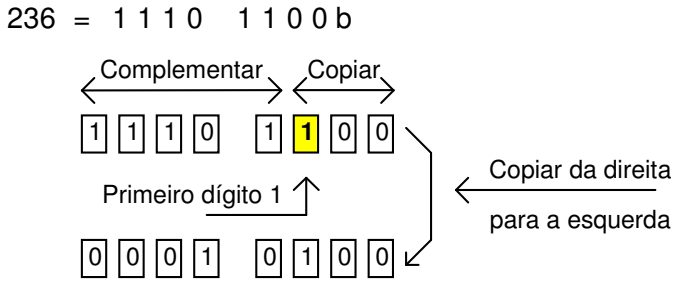

Figura A.8. Ilustração do algoritmo para calcular o complemento dois do número 236.

Quando o número está em hexadecimal o cálculo do complemento dois ainda é fácil. Basta calcular o complemento a 15 e depois somar 1.

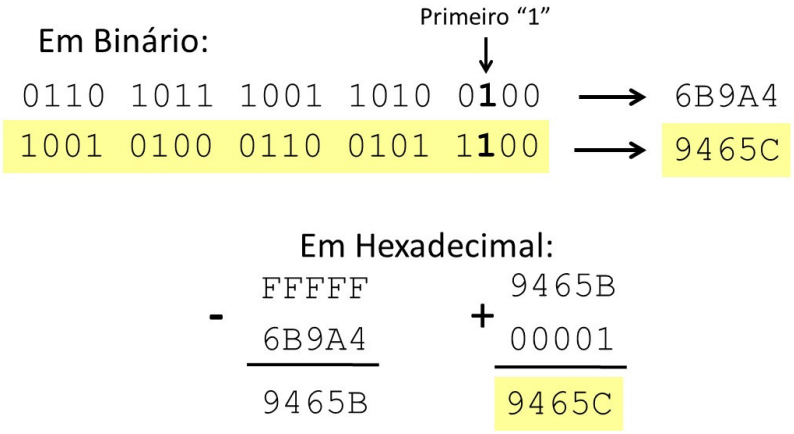

Figura A.9. Ilustração do algoritmo para calcular o complemento dois com o número representado em binário e em hexadecimal.

Um fato interessante é que as operações de soma e subtração independem da representação que se adote: com ou sem sinal. Por este fato, as operações aritméticas envolvendo números negativos em representação com complemento dois são muito simples, pois se houver "vai um" no último *bit*, ele simplesmente é ignorado. A Figura A.10 apresenta exemplos de operações.

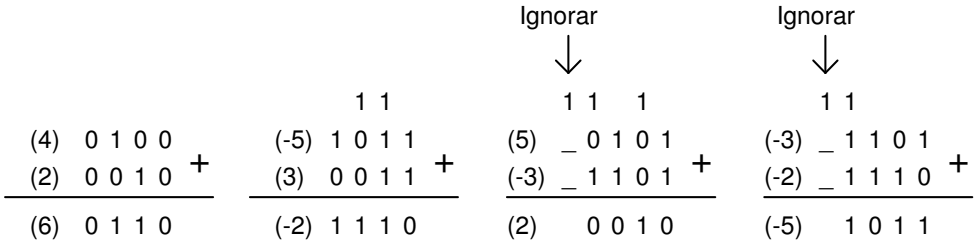

Figura A.10. Quatro somas envolvendo todas as combinações possíveis entre números positivos e negativos, representados com complemento dois.

É preciso ainda explicar conceito de **overflow**. Ao se trabalhar com um número fixo de *bits*, pode haver uma operação de soma que leve a um resultado que está além das possibilidades de representação daquele conjunto de *bits*.

Por exemplo, voltando ao caso de trabalhar com 4 *bits* em complemento dois, os números devem estar entre -8 e +7. Qualquer número aquém ou além desses limites acarretará num *overflow*. Para cada operação, é necessário um teste para verificar o *overflow*. Isto pode parecer complicado, mas ele é operacionalmente muito fácil de ser implementado. É lógico que a soma de dois números positivos deva dar como resultado um número positivo e que o resultado da soma de dois números negativos deva ser um número negativo. Se isto não acontecer, após desprezar-se o *carry*, é porque houve um *overflow*.

A Figura A.11 apresenta dois casos de *overflow*, em representação com 4 *bits*. Nesta figura, na operação da esquerda, a soma de dois números positivos 4 + 5 teve como resultado um número negativo (-7) e, na operação da direita, a soma de dois números negativos (-5) + (-6) teve como resultado um número positivo 5. É importante lembrar que o *overflow* surgiu porque se está usando uma representação limitada a 4 *bits*. Se a representação tivesse 8 *bits*, esses números não provocariam *overflow*. Porém este erro seria ocasionado por outro par de números. Para não confundir o *carry* com o *overflow* numa representação sem sinal, na soma, ambos são iguais e, na subtração, ambos são complementares.

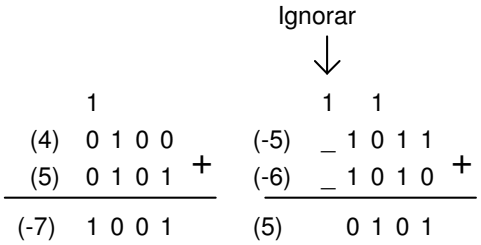

Figura A.11. Duas somas levando a overflow na representação com 4 bits e usando complemento dois.

Para finalizar vamos fazer explicar o que seria o equivalente ao complemento 2, usando a base 10. Por simplicidade vamos nos limitar a uma representação com apenas 2 dígitos decimais. Assim, os números válidos vão desde 00 até 99. A Figura A.12 apresenta algumas somas com o número 37, onde se ignorou o carry (vai um) e se sugere os números negativos em complemento a 10.

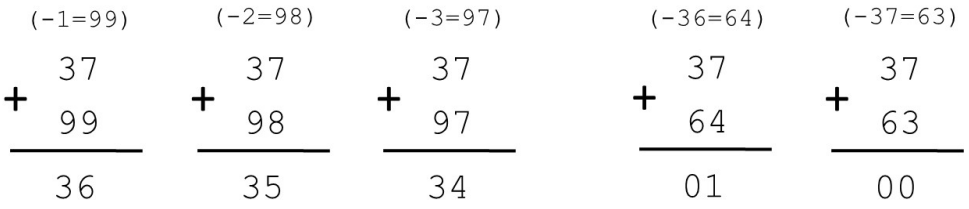

Figura A.12. Somas em base 10 ignorando o carry (vai um) para ilustrar a representação de números negativos usando 2 dígitos decimais.

No cálculo do complemento a 10, primeiro calculamos o complemento a 9 e depois somamos 1. O cálculo do número -36 em complemento a 10 está apresentado na Figura A.13.

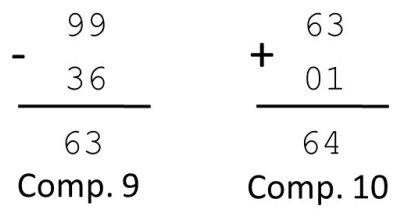

Figura A.13. Cálculo do Complemento a 10 do número 36.