CEP - UFPR - UP - UTFPR - PUCPr –
30/08/2024 - 11:15:46.2
Prof Dr P Kantek (pkantek@gmail.com)
Representação de números em máquina
VIVXq32a, V: 1.05 76506 CARLOS EDUARDO ANTAO 24CC2403 - 1 _____ / _____ / _____

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Agradeçamos a George Boole e a Claude Shannon por isso. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Ou seja, na matematica computacional estamos presos à matemática finita de Karl Gauss e aos grupos de Evariste Galois.

Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> dif=(((5**0.5)**2)-5)
>>> print(f'{dif:.24f}')
0.000000000000000888178420
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{lcl} 11111111 & = & -(2^6 + ... + 2^1 + 2^0) = -127 \\ \quad ... & & \\ 10000001 & = & \text{-1} \\ 10000000 & = & \text{-0 (?)} \\ 00000000 & = & 0 \\ \quad ... & & \\ 01111111 & = & 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e viceversa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 \cdots e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0$ e -0) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

Vamos ver etapa a etapa, como os resultados acima foram encontrados: Começando com o inteiro 1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1 (caso 1, acima). Como o primeiro bit é 1, o número é negativo e precisamos invertê-lo. Fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 e agora, soma-se 1 e fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1. A conversão deste número para base decimal é 1+2+8+16+128+1024+8192=9371. Portanto a resposta e -9371.

Tente achar por sua conta o segundo exemplo.

Quanto ao terceiro caso, o flutuante 1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0, devemos estudar a distribuição de bits. Vamos lá

```
1 | 1 0 0 1 1 | 0 0 0 1 0 0 0 0 1 0
s   e e e e e   m m m m m m m m m m
```

O sinal do número é obtido fazendo -1^s e neste caso o número será negativo.

O expoente é -15+19=4, portanto ao final ter-se-á que multiplicar a mantissa por $2^4 = 16$

Finalmente a mantissa, é o número 0 0 0 1 0 0 0 0 1 0, que deve ter um 1 colocado à frente dando o binário de 11 bits 1 0 0 0 1 0 0 0 0 1 0. Convertendo este número a decimal tem-se 2+64+1024=1090, que deve ser dividido por $2^{11} = 2048$ dando 1090/2048 = 0.5322265625 que multipilcado por 16 da $0.5322265625 \times 16 = 8.515625$ ou -8.515625.

Vamos ao quarto caso: 0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0 ou 0 | 1 1 0 0 0 | 0 1 1 1 0 0 1 0 1 0 O número é positivo, e seu expoente é -15+24=9. A mantissa é 1,0 1 1 1 0 0 1 0 1 0 = 1482. Ao final, o número é $+2^9 \times (1482/2048) = 370.5$.

Tente resolver agora o quinto e o sexto casos.

Como o sistema converte

Agora, a grande questão: quando você digita um número em um programa ou em uma tela, o computador precisa registrar este número na memória, usando um dos formatos aqui estudados. Vamos estudar o processo de conversão $decimal \rightarrow float$ e para campos de 16 bits. Vale a referência de este tamanho de campo não é mais usado, e só está aqui pela facilidade e simplicidade dos cálculos envolvidos. Vamos por etapas:

1. Dado um número real, por exemplo 183.27, a primeira coisa a fazer é converter este número em binário. Fica:
   $187.27_{(10)} \rightarrow 10111011.0100010100_{(2)}$

2. Agora, o ponto decimal deve ser levado à esquerda, até o primeiro 1 do número. A cada casa, o expoente deve ser incrementado. Acompanhe:
   $10111011.0100010100 \times 2^0$
   $1011101.10100010100 \times 2^1$
   $101110.110100010100 \times 2^2$
   $10111.0110100010100 \times 2^3$
   $1011.10110100010100 \times 2^4$
   $101.110110100010100 \times 2^5$
   $10.1110110100010100 \times 2^6$
   $1.01110110100010100 \times 2^7$
   $.101110110100010100 \times 2^8$
   Note que as 7 linhas acima representam o mesmo número.

3. Como o número é positivo, seu sinal é 0.

4. A potência do número é 8 que deve ser somado com 15, totalizando 23. Agora, é hora de converter o expoente em binário: $23_{(10)} \rightarrow 10111_{(2)}$

5. Finalmente, e hora de pegar 10 bits da mantissa, lembrando que o primeiro 1 (aquele que fica depois do ponto) é omitido. A mantissa fica: 0111011010

6. o numero binario em formato binary16 é
   `0 | 1 0 1 1 1 | 0 1 1 1 0 1 1 0 1 0`

7. Se você converter este número para decimal, vai encontrar 187.25, o que demonstra a baixa precisão deste formato (e explica porque ele não é usado).

☞ Para você fazer

Interprete os 2 números binários a seguir como *binary*16 inteiros e negativos e ache os valores decimais
1 e 2.

```
1 1 1 0 1 0 1 0 1 1 0 0 1 1 0 0
1 1 1 1 1 1 1 1 1 0 0 1 1 1 0 0 0
```

e agora interprete os 4 números a seguir como *binary*16 flutuantes e ache os decimais equivalentes com 5 casas decimais
3, 4, 5 e 6.

```
1 0 1 1 0 1 0 1 1 1 1 1 0 1 0 1 1
0 1 1 1 0 0 0 0 1 0 1 1 0 0 1 1
1 1 0 1 0 1 0 1 1 0 0 0 0 0 0 0
1 1 1 1 0 0 1 0 0 0 0 1 0 0 1 0
```

Finalmente, converta os valores abaixo que estão em decimal para o formato *binary*16
7 e 8.

```
99
-56.66
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.
7			
8			

CEP - UFPR - UP - UTFPR - PUCPr –
30/08/2024 - 11:15:46.2
Prof Dr P Kantek (pkantek@gmail.com)
Representação de números em máquina
VIVXq32a, V: 1.05 76663 DANILO DEPETRIS SOARES 24CC2403 - 2 _____ / _____ / _____

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Agradeçamos a George Boole e a Claude Shannon por isso. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Ou seja, na matematica computacional estamos presos à matemática finita de Karl Gauss e aos grupos de Evariste Galois.

Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> dif=(((5**0.5)**2)-5)
>>> print(f'{dif:.24f}')
0.000000000000000888178420
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{lcl} 11111111 & = & -(2^6 + ... + 2^1 + 2^0) = -127 \\ \quad ... & & \\ 10000001 & = & \text{-1} \\ 10000000 & = & \text{-0 (?)} \\ 00000000 & = & 0 \\ \quad ... & & \\ 01111111 & = & 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e viceversa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 \cdots e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0$ e -0) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

Vamos ver etapa a etapa, como os resultados acima foram encontrados: Começando com o inteiro 1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1 (caso 1, acima). Como o primeiro bit é 1, o número é negativo e precisamos invertê-lo. Fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 e agora, soma-se 1 e fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1. A conversão deste número para base decimal é 1+2+8+16+128+1024+8192=9371. Portanto a resposta e -9371.

Tente achar por sua conta o segundo exemplo.

Quanto ao terceiro caso, o flutuante 1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0, devemos estudar a distribuição de bits. Vamos lá

```
1 | 1 0 0 1 1 | 0 0 0 1 0 0 0 0 1 0
s   e e e e e   m m m m m m m m m m
```

O sinal do número é obtido fazendo -1^s e neste caso o número será negativo.

O expoente é -15+19=4, portanto ao final ter-se-á que multiplicar a mantissa por $2^4 = 16$

Finalmente a mantissa, é o número 0 0 0 1 0 0 0 0 1 0, que deve ter um 1 colocado à frente dando o binário de 11 bits 1 0 0 0 1 0 0 0 0 1 0. Convertendo este número a decimal tem-se 2+64+1024=1090, que deve ser dividido por $2^{11} = 2048$ dando $1090/2048 = 0.5322265625$ que multipilcado por 16 da $0.5322265625 \times 16 = 8.515625$ ou -8.515625.

Vamos ao quarto caso: 0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0 ou 0 | 1 1 0 0 0 | 0 1 1 1 0 0 1 0 1 0 O número é positivo, e seu expoente é -15+24=9. A mantissa é 1,0 1 1 1 0 0 1 0 1 0 = 1482. Ao final, o número é $+2^9 \times (1482/2048) = 370.5$.

Tente resolver agora o quinto e o sexto casos.

Como o sistema converte

Agora, a grande questão: quando você digita um número em um programa ou em uma tela, o computador precisa registrar este número na memória, usando um dos formatos aqui estudados. Vamos estudar o processo de conversão $decimal \rightarrow float$ e para campos de 16 bits. Vale a referência de este tamanho de campo não é mais usado, e só está aqui pela facilidade e simplicidade dos cálculos envolvidos. Vamos por etapas:

1. Dado um número real, por exemplo 183.27, a primeira coisa a fazer é converter este número em binário. Fica:
   $187.27_{(10)} \rightarrow 10111011.0100010100_{(2)}$

2. Agora, o ponto decimal deve ser levado à esquerda, até o primeiro 1 do número. A cada casa, o expoente deve ser incrementado. Acompanhe:
   $10111011.0100010100 \times 2^0$
   $1011101.10100010100 \times 2^1$
   $101110.110100010100 \times 2^2$
   $10111.0110100010100 \times 2^3$
   $1011.10110100010100 \times 2^4$
   $101.110110100010100 \times 2^5$
   $10.1110110100010100 \times 2^6$
   $1.01110110100010100 \times 2^7$
   $.101110110100010100 \times 2^8$
   Note que as 7 linhas acima representam o mesmo número.

3. Como o número é positivo, seu sinal é 0.

4. A potência do número é 8 que deve ser somado com 15, totalizando 23. Agora, é hora de converter o expoente em binário: $23_{(10)} \rightarrow 10111_{(2)}$

5. Finalmente, e hora de pegar 10 bits da mantissa, lembrando que o primeiro 1 (aquele que fica depois do ponto) é omitido. A mantissa fica: 0111011010

6. o numero binario em formato binary16 é
   `0 | 1 0 1 1 1 | 0 1 1 1 0 1 1 0 1 0`

7. Se você converter este número para decimal, vai encontrar 187.25, o que demonstra a baixa precisão deste formato (e explica porque ele não é usado).

☞ Para você fazer

Interprete os 2 números binários a seguir como *binary*16 inteiros e negativos e ache os valores decimais
1 e 2.

```
1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0
1 1 1 1 1 0 0 1 1 1 1 1 0 0 1 1 0
```

e agora interprete os 4 números a seguir como *binary*16 flutuantes e ache os decimais equivalentes com 5 casas decimais
3, 4, 5 e 6.

```
0 1 1 0 0 1 0 0 0 1 1 0 1 0 1 0
0 0 0 0 0 1 1 1 0 1 0 0 0 1 0 1
1 1 0 0 1 1 0 0 1 0 1 1 0 0 1 1
0 0 1 1 1 0 1 1 1 1 1 1 1 1 0 0
```

Finalmente, converta os valores abaixo que estão em decimal para o formato *binary*16
7 e 8.

```
115
-53.64
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.
7			
8			

CEP - UFPR - UP - UTFPR - PUCPr – 30/08/2024 - 11:15:46.2
Prof Dr P Kantek (pkantek@gmail.com)
Representação de números em máquina
VIVXq32a, V: 1.05 76513 EDUARDO CAVALHEIRO DIAS 24CC2403 - 3 _____ / / _____

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Agradeçamos a George Boole e a Claude Shannon por isso. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Ou seja, na matematica computacional estamos presos à matemática finita de Karl Gauss e aos grupos de Evariste Galois.

Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> dif=(((5**0.5)**2)-5)
>>> print(f'{dif:.24f}')
0.000000000000000888178420
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{lcl} 11111111 & = & -(2^6 + ... + 2^1 + 2^0) = -127 \\ \ldots & & \\ 10000001 & = & \text{-1} \\ 10000000 & = & \text{-0 (?)} \\ 00000000 & = & 0 \\ \ldots & & \\ 01111111 & = & 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e viceversa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 ... e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0$ e -0) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

Vamos ver etapa a etapa, como os resultados acima foram encontrados: Começando com o inteiro 1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1 (caso 1, acima). Como o primeiro bit é 1, o número é negativo e precisamos invertê-lo. Fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 e agora, soma-se 1 e fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1. A conversão deste número para base decimal é 1+2+8+16+128+1024+8192=9371. Portanto a resposta e -9371.

Tente achar por sua conta o segundo exemplo.

Quanto ao terceiro caso, o flutuante 1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0, devemos estudar a distribuição de bits. Vamos lá

```
1 | 1 0 0 1 1 | 0 0 0 1 0 0 0 0 1 0
s   e e e e e   m m m m m m m m m m
```

O sinal do número é obtido fazendo -1^s e neste caso o número será negativo.

O expoente é -15+19=4, portanto ao final ter-se-á que multiplicar a mantissa por $2^4 = 16$

Finalmente a mantissa, é o número 0 0 0 1 0 0 0 0 1 0, que deve ter um 1 colocado à frente dando o binário de 11 bits 1 0 0 0 1 0 0 0 0 1 0. Convertendo este número a decimal tem-se 2+64+1024=1090, que deve ser dividido por $2^{11} = 2048$ dando $1090/2048 = 0.5322265625$ que multipilcado por 16 da $0.5322265625 \times 16 = 8.515625$ ou -8.515625.

Vamos ao quarto caso: 0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0 ou `0 | 1 1 0 0 0 | 0 1 1 1 0 0 1 0 1 0` O número é positivo, e seu expoente é -15+24=9. A mantissa é 1,0 1 1 1 0 0 1 0 1 0 = 1482. Ao final, o número é $+2^9 \times (1482/2048) = 370.5$.

Tente resolver agora o quinto e o sexto casos.

Como o sistema converte

Agora, a grande questão: quando você digita um número em um programa ou em uma tela, o computador precisa registrar este número na memória, usando um dos formatos aqui estudados. Vamos estudar o processo de conversão $decimal \rightarrow float$ e para campos de 16 bits. Vale a referência de este tamanho de campo não é mais usado, e só está aqui pela facilidade e simplicidade dos cálculos envolvidos. Vamos por etapas:

1. Dado um número real, por exemplo 183.27, a primeira coisa a fazer é converter este número em binário. Fica:
   $187.27_{(10)} \rightarrow 10111011.0100010100_{(2)}$

2. Agora, o ponto decimal deve ser levado à esquerda, até o primeiro 1 do número. A cada casa, o expoente deve ser incrementado. Acompanhe:
   $10111011.0100010100 \times 2^0$
   $1011101.10100010100 \times 2^1$
   $101110.110100010100 \times 2^2$
   $10111.0110100010100 \times 2^3$
   $1011.10110100010100 \times 2^4$
   $101.110110100010100 \times 2^5$
   $10.1110110100010100 \times 2^6$
   $1.01110110100010100 \times 2^7$
   $.10110110100010100 \times 2^8$
   Note que as 7 linhas acima representam o mesmo número.

3. Como o número é positivo, seu sinal é 0.

4. A potência do número é 8 que deve ser somado com 15, totalizando 23. Agora, é hora de converter o expoente em binário: $23_{(10)} \rightarrow 10111_{(2)}$

5. Finalmente, e hora de pegar 10 bits da mantissa, lembrando que o primeiro 1 (aquele que fica depois do ponto) é omitido. A mantissa fica: 0111011010

6. o numero binario em formato binary16 é
   `0 | 1 0 1 1 1 | 0 1 1 1 0 1 1 0 1 0`

7. Se você converter este número para decimal, vai encontrar 187.25, o que demonstra a baixa precisão deste formato (e explica porque ele não é usado).

☞ Para você fazer

Interprete os 2 números binários a seguir como *binary*16 inteiros e negativos e ache os valores decimais
1 e 2.

```
1 1 1 1 1 0 0 1 1 0 0 0 0 1 0 0
1 1 1 1 0 0 1 0 0 0 0 0 0 1 0 0
```

e agora interprete os 4 números a seguir como *binary*16 flutuantes e ache os decimais equivalentes com 5 casas decimais
3, 4, 5 e 6.

```
1 0 0 0 1 0 0 0 1 0 0 1 1 1 1 0
1 1 0 1 1 1 1 0 1 1 1 0 0 0 1 0
0 0 0 1 0 0 0 1 0 0 1 0 0 0 0 1
1 1 1 0 0 0 1 1 0 1 0 1 1 0 1 1
```

Finalmente, converta os valores abaixo que estão em decimal para o formato *binary*16
7 e 8.

```
125
-53.26
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.
7			
8			

CEP - UFPR - UP - UTFPR - PUCPr –
30/08/2024 - 11:15:46.2
Prof Dr P Kantek (pkantek@gmail.com)
Representação de números em máquina
VIVXq32a, V: 1.05 76520 ERIC CORDEIRO
24CC2403 - 4 _____ / _____ / _____

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Agradeçamos a George Boole e a Claude Shannon por isso. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Ou seja, na matematica computacional estamos presos à matemática finita de Karl Gauss e aos grupos de Evariste Galois.

Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> dif=(((5**0.5)**2)-5)
>>> print(f'{dif:.24f}')
0.000000000000000888178420
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{lcl} 11111111 & = & -(2^6 + ... + 2^1 + 2^0) = -127 \\ \quad ... & & \\ 10000001 & = & \text{-1} \\ 10000000 & = & \text{-0 (?)} \\ 00000000 & = & 0 \\ \quad ... & & \\ 01111111 & = & 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e viceversa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 \cdots e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0$ e -0) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

Vamos ver etapa a etapa, como os resultados acima foram encontrados: Começando com o inteiro 1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1 (caso 1, acima). Como o primeiro bit é 1, o número é negativo e precisamos invertê-lo. Fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 e agora, soma-se 1 e fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1. A conversão deste número para base decimal é 1+2+8+16+128+1024+8192=9371. Portanto a resposta e -9371.

Tente achar por sua conta o segundo exemplo.

Quanto ao terceiro caso, o flutuante 1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0, devemos estudar a distribuição de bits. Vamos lá

```
1 | 1 0 0 1 1 | 0 0 0 1 0 0 0 0 1 0
s   e e e e e   m m m m m m m m m m
```

O sinal do número é obtido fazendo -1^s e neste caso o número será negativo.

O expoente é -15+19=4, portanto ao final ter-se-á que multiplicar a mantissa por $2^4 = 16$

Finalmente a mantissa, é o número 0 0 0 1 0 0 0 0 0 1 0, que deve ter um 1 colocado à frente dando o binário de 11 bits 1 0 0 0 1 0 0 0 0 1 0. Convertendo este número a decimal tem-se 2+64+1024=1090, que deve ser dividido por $2^{11} = 2048$ dando $1090/2048 = 0.5322265625$ que multipilcado por 16 da $0.5322265625 \times 16 = 8.515625$ ou -8.515625.

Vamos ao quarto caso: 0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0 ou 0 | 1 1 0 0 0 | 0 1 1 1 0 0 1 0 1 0 O número é positivo, e seu expoente é -15+24=9. A mantissa é 1,0 1 1 1 0 0 1 0 1 0 = 1482. Ao final, o número é $+2^9 \times (1482/2048) = 370.5$.

Tente resolver agora o quinto e o sexto casos.

Como o sistema converte

Agora, a grande questão: quando você digita um número em um programa ou em uma tela, o computador precisa registrar este número na memória, usando um dos formatos aqui estudados. Vamos estudar o processo de conversão $decimal \rightarrow float$ e para campos de 16 bits. Vale a referência de este tamanho de campo não é mais usado, e só está aqui pela facilidade e simplicidade dos cálculos envolvidos. Vamos por etapas:

1. Dado um número real, por exemplo 183.27, a primeira coisa a fazer é converter este número em binário. Fica:
   $187.27_{(10)} \rightarrow 10111011.0100010100_{(2)}$

2. Agora, o ponto decimal deve ser levado à esquerda, até o primeiro 1 do número. A cada casa, o expoente deve ser incrementado. Acompanhe:
   $10111011.0100010100 \times 2^0$
   $1011101.10100010100 \times 2^1$
   $101110.110100010100 \times 2^2$
   $10111.0110100010100 \times 2^3$
   $1011.10110100010100 \times 2^4$
   $101.110110100010100 \times 2^5$
   $10.1110110100010100 \times 2^6$
   $1.01110110100010100 \times 2^7$
   $.10110110100010100 \times 2^8$
   Note que as 7 linhas acima representam o mesmo número.

3. Como o número é positivo, seu sinal é 0.

4. A potência do número é 8 que deve ser somado com 15, totalizando 23. Agora, é hora de converter o expoente em binário: $23_{(10)} \rightarrow 10111_{(2)}$

5. Finalmente, e hora de pegar 10 bits da mantissa, lembrando que o primeiro 1 (aquele que fica depois do ponto) é omitido. A mantissa fica: 0111011010

6. o numero binario em formato binary16 é
   `0 | 1 0 1 1 1 | 0 1 1 1 0 1 1 0 1 0`

7. Se você converter este número para decimal, vai encontrar 187.25, o que demonstra a baixa precisão deste formato (e explica porque ele não é usado).

☞ Para você fazer

Interprete os 2 números binários a seguir como $binary16$ inteiros e negativos e ache os valores decimais
1 e 2.

```
1 1 1 0 1 1 0 1 0 1 1 0 1 0 0 0
1 1 1 0 1 1 0 1 1 1 1 1 1 0 1 0
```

e agora interprete os 4 números a seguir como $binary16$ flutuantes e ache os decimais equivalentes com 5 casas decimais
3, 4, 5 e 6.

```
0 0 0 1 0 0 0 0 1 1 0 1 1 1 0 0
0 1 0 1 1 0 0 0 1 1 0 1 1 0 1 1
0 1 0 1 0 0 1 0 1 0 0 1 0 0 1 1
1 1 0 0 0 0 0 0 0 1 0 0 0 1 1 1
```

Finalmente, converta os valores abaixo que estão em decimal para o formato $binary16$
7 e 8.

```
97
-45.48
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.
7			
8			

CEP - UFPR - UP - UTFPR - PUCPr –
30/08/2024 - 11:15:46.2
Prof Dr P Kantek (pkantek@gmail.com)
Representação de números em máquina
VIVXq32a, V: 1.05 76537 FELIPE EIJI KANO
24CC2403 - 5 _____ / _____ / _____

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Agradeçamos a George Boole e a Claude Shannon por isso. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Ou seja, na matematica computacional estamos presos à matemática finita de Karl Gauss e aos grupos de Evariste Galois.

Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> dif=(((5**0.5)**2)-5)
>>> print(f'{dif:.24f}')
0.000000000000000888178420
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{lcl} 11111111 & = & -(2^6 + ... + 2^1 + 2^0) = -127 \\ \ldots & & \\ 10000001 & = & \text{-1} \\ 10000000 & = & \text{-0 (?)} \\ 00000000 & = & 0 \\ \ldots & & \\ 01111111 & = & 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e viceversa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 \cdots e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0$ e -0) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

Vamos ver etapa a etapa, como os resultados acima foram encontrados: Começando com o inteiro 1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1 (caso 1, acima). Como o primeiro bit é 1, o número é negativo e precisamos invertê-lo. Fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 e agora, soma-se 1 e fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1. A conversão deste número para base decimal é 1+2+8+16+128+1024+8192=9371. Portanto a resposta e -9371.

Tente achar por sua conta o segundo exemplo.

Quanto ao terceiro caso, o flutuante 1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0, devemos estudar a distribuição de bits. Vamos lá

```
1 | 1 0 0 1 1 | 0 0 0 1 0 0 0 0 1 0
s   e e e e e   m m m m m m m m m m
```

O sinal do número é obtido fazendo -1^s e neste caso o número será negativo.

O expoente é -15+19=4, portanto ao final ter-se-á que multiplicar a mantissa por $2^4 = 16$

Finalmente a mantissa, é o número 0 0 0 1 0 0 0 0 1 0, que deve ter um 1 colocado à frente dando o binário de 11 bits 1 0 0 0 1 0 0 0 0 1 0. Convertendo este número a decimal tem-se 2+64+1024=1090, que deve ser dividido por $2^{11} = 2048$ dando $1090/2048 = 0.5322265625$ que multipilcado por 16 da $0.5322265625 \times 16 = 8.515625$ ou -8.515625.

Vamos ao quarto caso: 0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0 ou 0 | 1 1 0 0 0 | 0 1 1 1 0 0 1 0 1 0 O número é positivo, e seu expoente é -15+24=9. A mantissa é 1,0 1 1 1 0 0 1 0 1 0 = 1482. Ao final, o número é $+2^9 \times (1482/2048) = 370.5$.

Tente resolver agora o quinto e o sexto casos.

Como o sistema converte

Agora, a grande questão: quando você digita um número em um programa ou em uma tela, o computador precisa registrar este número na memória, usando um dos formatos aqui estudados. Vamos estudar o processo de conversão $decimal \rightarrow float$ e para campos de 16 bits. Vale a referência de este tamanho de campo não é mais usado, e só está aqui pela facilidade e simplicidade dos cálculos envolvidos. Vamos por etapas:

1. Dado um número real, por exemplo 183.27, a primeira coisa a fazer é converter este número em binário. Fica:
   $187.27_{(10)} \rightarrow 10111011.0100010100_{(2)}$

2. Agora, o ponto decimal deve ser levado à esquerda, até o primeiro 1 do número. A cada casa, o expoente deve ser incrementado. Acompanhe:
   $10111011.0100010100 \times 2^0$
   $1011101.10100010100 \times 2^1$
   $101110.110100010100 \times 2^2$
   $10111.0110100010100 \times 2^3$
   $1011.10110100010100 \times 2^4$
   $101.110110100010100 \times 2^5$
   $10.1110110100010100 \times 2^6$
   $1.01110110100010100 \times 2^7$
   $.101110110100010100 \times 2^8$
   Note que as 7 linhas acima representam o mesmo número.

3. Como o número é positivo, seu sinal é 0.

4. A potência do número é 8 que deve ser somado com 15, totalizando 23. Agora, é hora de converter o expoente em binário: $23_{(10)} \rightarrow 10111_{(2)}$

5. Finalmente, e hora de pegar 10 bits da mantissa, lembrando que o primeiro 1 (aquele que fica depois do ponto) é omitido. A mantissa fica: 0111011010

6. o numero binario em formato binary16 é
   `0 | 1 0 1 1 1 | 0 1 1 1 0 1 1 0 1 0`

7. Se você converter este número para decimal, vai encontrar 187.25, o que demonstra a baixa precisão deste formato (e explica porque ele não é usado).

☞ Para você fazer

Interprete os 2 números binários a seguir como $binary16$ inteiros e negativos e ache os valores decimais
1 e 2.

```
1 1 1 1 1 0 0 1 1 1 1 0 1 1 0 0
1 1 1 1 1 0 1 0 0 1 0 1 0 0 1 0
```

e agora interprete os 4 números a seguir como $binary16$ flutuantes e ache os decimais equivalentes com 5 casas decimais
3, 4, 5 e 6.

```
0 1 1 1 0 0 1 1 1 0 1 1 1 0 0 1
1 0 1 0 0 1 1 1 1 0 0 1 0 1 0 1
0 0 0 0 0 0 1 1 0 1 0 0 1 0 0 1
0 0 1 0 0 1 1 1 1 0 0 1 0 1 0 0
```

Finalmente, converta os valores abaixo que estão em decimal para o formato $binary16$
7 e 8.

```
107
-40.06
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.
7			
8			

CEP - UFPR - UP - UTFPR - PUCPr –
30/08/2024 - 11:15:46.2
Prof Dr P Kantek (pkantek@gmail.com)
Representação de números em máquina
VIVXq32a, V: 1.05 76544 FELIPE JESMIEL
LEITE 24CC2403 - 6 _____ / _____ / _____

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Agradeçamos a George Boole e a Claude Shannon por isso. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Ou seja, na matematica computacional estamos presos à matemática finita de Karl Gauss e aos grupos de Evariste Galois.

Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> dif=(((5**0.5)**2)-5)
>>> print(f'{dif:.24f}')
0.000000000000000888178420
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{lcl} 11111111 & = & -(2^6 + ... + 2^1 + 2^0) = -127 \\ \ldots & & \\ 10000001 & = & \text{-1} \\ 10000000 & = & \text{-0 (?)} \\ 00000000 & = & 0 \\ \ldots & & \\ 01111111 & = & 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e viceversa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 \cdots e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0$ e -0) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

Vamos ver etapa a etapa, como os resultados acima foram encontrados: Começando com o inteiro 1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1 (caso 1, acima). Como o primeiro bit é 1, o número é negativo e precisamos invertê-lo. Fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 e agora, soma-se 1 e fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1. A conversão deste número para base decimal é 1+2+8+16+128+1024+8192=9371. Portanto a resposta e -9371.

Tente achar por sua conta o segundo exemplo.

Quanto ao terceiro caso, o flutuante 1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0, devemos estudar a distribuição de bits. Vamos lá

```
1 | 1 0 0 1 1 | 0 0 0 1 0 0 0 0 1 0
s   e e e e e   m m m m m m m m m m
```

O sinal do número é obtido fazendo -1^s e neste caso o número será negativo.

O expoente é -15+19=4, portanto ao final ter-se-á que multiplicar a mantissa por $2^4 = 16$

Finalmente a mantissa, é o número 0 0 0 1 0 0 0 0 1 0, que deve ter um 1 colocado à frente dando o binário de 11 bits 1 0 0 0 1 0 0 0 0 1 0. Convertendo este número a decimal tem-se 2+64+1024=1090, que deve ser dividido por $2^{11} = 2048$ dando $1090/2048 = 0.5322265625$ que multipilcado por 16 da $0.5322265625 \times 16 = 8.515625$ ou -8.515625.

Vamos ao quarto caso: 0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0 ou 0 | 1 1 0 0 0 | 0 1 1 1 0 0 1 0 1 0 O número é positivo, e seu expoente é -15+24=9. A mantissa é 1,0 1 1 1 0 0 1 0 1 0 = 1482. Ao final, o número é $+2^9 \times (1482/2048) = 370.5$.

Tente resolver agora o quinto e o sexto casos.

Como o sistema converte

Agora, a grande questão: quando você digita um número em um programa ou em uma tela, o computador precisa registrar este número na memória, usando um dos formatos aqui estudados. Vamos estudar o processo de conversão $decimal \rightarrow float$ e para campos de 16 bits. Vale a referência de este tamanho de campo não é mais usado, e só está aqui pela facilidade e simplicidade dos cálculos envolvidos. Vamos por etapas:

1. Dado um número real, por exemplo 183.27, a primeira coisa a fazer é converter este número em binário. Fica:
   $187.27_{(10)} \rightarrow 10111011.0100010100_{(2)}$

2. Agora, o ponto decimal deve ser levado à esquerda, até o primeiro 1 do número. A cada casa, o expoente deve ser incrementado. Acompanhe:
   $10111011.0100010100 \times 2^0$
   $1011101.10100010100 \times 2^1$
   $101110.110100010100 \times 2^2$
   $10111.0110100010100 \times 2^3$
   $1011.10110100010100 \times 2^4$
   $101.110110100010100 \times 2^5$
   $10.1110110100010100 \times 2^6$
   $1.01110110100010100 \times 2^7$
   $.101110110100010100 \times 2^8$
   Note que as 7 linhas acima representam o mesmo número.

3. Como o número é positivo, seu sinal é 0.

4. A potência do número é 8 que deve ser somado com 15, totalizando 23. Agora, é hora de converter o expoente em binário: $23_{(10)} \rightarrow 10111_{(2)}$

5. Finalmente, e hora de pegar 10 bits da mantissa, lembrando que o primeiro 1 (aquele que fica depois do ponto) é omitido. A mantissa fica: 0111011010

6. o numero binario em formato binary16 é
   `0 | 1 0 1 1 1 | 0 1 1 1 0 1 1 0 1 0`

7. Se você converter este número para decimal, vai encontrar 187.25, o que demonstra a baixa precisão deste formato (e explica porque ele não é usado).

☞ Para você fazer

Interprete os 2 números binários a seguir como $binary16$ inteiros e negativos e ache os valores decimais
1 e 2.

```
1 1 1 0 1 0 1 1 1 0 0 1 0 1 0 1
1 1 0 1 1 1 0 0 0 1 0 1 1 0 0 0
```

e agora interprete os 4 números a seguir como $binary16$ flutuantes e ache os decimais equivalentes com 5 casas decimais
3, 4, 5 e 6.

```
0 0 1 0 1 1 0 0 1 1 0 1 0 0 0 1
0 1 1 1 0 1 0 0 0 1 1 1 1 0 0 1
0 0 0 0 0 0 1 1 1 0 1 1 1 0 0 1
0 0 1 1 1 0 0 1 0 0 0 0 1 0 1 0
```

Finalmente, converta os valores abaixo que estão em decimal para o formato $binary16$
7 e 8.

```
116
-44.36
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.
7			
8			

CEP - UFPR - UP - UTFPR - PUCPr –
30/08/2024 - 11:15:46.2
Prof Dr P Kantek (pkantek@gmail.com)
Representação de números em máquina
VIVXq32a, V: 1.05 76551 GABRIELA BERNARDO PASQUALOTTO 24CC2403 - 7
/ _____ / _____

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Agradeçamos a George Boole e a Claude Shannon por isso. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Ou seja, na matematica computacional estamos presos à matemática finita de Karl Gauss e aos grupos de Evariste Galois.

Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> dif=(((5**0.5)**2)-5)
>>> print(f'{dif:.24f}')
0.000000000000000888178420
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{lcl} 11111111 & = & -(2^6 + ... + 2^1 + 2^0) = -127 \\ \quad ... & & \\ 10000001 & = & \text{-1} \\ 10000000 & = & \text{-0 (?)} \\ 00000000 & = & 0 \\ \quad ... & & \\ 01111111 & = & 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e viceversa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 ... e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0$ e -0) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

Vamos ver etapa a etapa, como os resultados acima foram encontrados: Começando com o inteiro 1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1 (caso 1, acima). Como o primeiro bit é 1, o número é negativo e precisamos invertê-lo. Fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 e agora, soma-se 1 e fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1. A conversão deste número para base decimal é 1+2+8+16+128+1024+8192=9371. Portanto a resposta e -9371.

Tente achar por sua conta o segundo exemplo.

Quanto ao terceiro caso, o flutuante 1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0, devemos estudar a distribuição de bits. Vamos lá

```
1 | 1 0 0 1 1 | 0 0 0 1 0 0 0 0 1 0
s   e e e e e   m m m m m m m m m m
```

O sinal do número é obtido fazendo -1^s e neste caso o número será negativo.

O expoente é -15+19=4, portanto ao final ter-se-á que multiplicar a mantissa por $2^4 = 16$

Finalmente a mantissa, é o número 0 0 0 1 0 0 0 0 1 0, que deve ter um 1 colocado à frente dando o binário de 11 bits 1 0 0 0 1 0 0 0 0 1 0. Convertendo este número a decimal tem-se 2+64+1024=1090, que deve ser dividido por $2^{11} = 2048$ dando $1090/2048 = 0.5322265625$ que multipilcado por 16 da $0.5322265625 \times 16 = 8.515625$ ou -8.515625.

Vamos ao quarto caso: 0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0 ou `0 | 1 1 0 0 0 | 0 1 1 1 0 0 1 0 1 0` O número é positivo, e seu expoente é -15+24=9. A mantissa é 1,0 1 1 1 0 0 1 0 1 0 = 1482. Ao final, o número é $+2^9 \times (1482/2048) = 370.5$.

Tente resolver agora o quinto e o sexto casos.

Como o sistema converte

Agora, a grande questão: quando você digita um número em um programa ou em uma tela, o computador precisa registrar este número na memória, usando um dos formatos aqui estudados. Vamos estudar o processo de conversão $decimal \rightarrow float$ e para campos de 16 bits. Vale a referência de este tamanho de campo não é mais usado, e só está aqui pela facilidade e simplicidade dos cálculos envolvidos. Vamos por etapas:

1. Dado um número real, por exemplo 183.27, a primeira coisa a fazer é converter este número em binário. Fica:
   $187.27_{(10)} \rightarrow 10111011.0100010100_{(2)}$

2. Agora, o ponto decimal deve ser levado à esquerda, até o primeiro 1 do número. A cada casa, o expoente deve ser incrementado. Acompanhe:
   $10111011.0100010100 \times 2^0$
   $1011101.10100010100 \times 2^1$
   $101110.110100010100 \times 2^2$
   $10111.0110100010100 \times 2^3$
   $1011.10110100010100 \times 2^4$
   $101.110110100010100 \times 2^5$
   $10.1110110100010100 \times 2^6$
   $1.01110110100010100 \times 2^7$
   $.101110110100010100 \times 2^8$
   Note que as 7 linhas acima representam o mesmo número.

3. Como o número é positivo, seu sinal é 0.

4. A potência do número é 8 que deve ser somado com 15, totalizando 23. Agora, é hora de converter o expoente em binário: $23_{(10)} \rightarrow 10111_{(2)}$

5. Finalmente, e hora de pegar 10 bits da mantissa, lembrando que o primeiro 1 (aquele que fica depois do ponto) é omitido. A mantissa fica: 0111011010

6. o numero binario em formato binary16 é
   `0 | 1 0 1 1 1 | 0 1 1 1 0 1 1 0 1 0`

7. Se você converter este número para decimal, vai encontrar 187.25, o que demonstra a baixa precisão deste formato (e explica porque ele não é usado).

☞ Para você fazer

Interprete os 2 números binários a seguir como *binary*16 inteiros e negativos e ache os valores decimais
1 e 2.

```
1 1 1 1 1 1 1 0 0 1 0 0 0 1 1 1
1 1 1 1 0 1 0 1 0 1 1 0 0 1 1 0
```

e agora interprete os 4 números a seguir como *binary*16 flutuantes e ache os decimais equivalentes com 5 casas decimais
3, 4, 5 e 6.

```
0 1 1 0 1 1 1 1 1 1 0 0 1 1 0 0
1 0 0 1 0 0 1 0 0 1 0 1 1 1 0 0
1 0 1 1 0 1 1 0 0 0 0 1 0 0 0 0
0 0 1 1 1 1 0 1 0 0 1 0 1 1 0 1
```

Finalmente, converta os valores abaixo que estão em decimal para o formato *binary*16
7 e 8.

```
135
-43.16
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.
7			
8			

CEP - UFPR - UP - UTFPR - PUCPr –
30/08/2024 - 11:15:46.2
Prof Dr P Kantek (pkantek@gmail.com)
Representação de números em máquina
VIVXq32a, V: 1.05 76568 GUILHERME VITOR DA SILVA 24CC2403 - 8 _____ / _____ / _____

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Agradeçamos a George Boole e a Claude Shannon por isso. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Ou seja, na matematica computacional estamos presos à matemática finita de Karl Gauss e aos grupos de Evariste Galois.

Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> dif=(((5**0.5)**2)-5)
>>> print(f'{dif:.24f}')
0.000000000000000888178420
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{ll} 11111111 & = -(2^6 + ... + 2^1 + 2^0) = -127 \\ \quad ... & \\ 10000001 & = \text{-1} \\ 10000000 & = \text{-0 (?)} \\ 00000000 & = 0 \\ \quad ... & \\ 01111111 & = 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e viceversa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 \cdots e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0$ e -0) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

Vamos ver etapa a etapa, como os resultados acima foram encontrados: Começando com o inteiro 1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1 (caso 1, acima). Como o primeiro bit é 1, o número é negativo e precisamos invertê-lo. Fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 e agora, soma-se 1 e fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1. A conversão deste número para base decimal é 1+2+8+16+128+1024+8192=9371. Portanto a resposta e -9371.

Tente achar por sua conta o segundo exemplo.

Quanto ao terceiro caso, o flutuante 1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0, devemos estudar a distribuição de bits. Vamos lá

```
1 | 1 0 0 1 1 | 0 0 0 1 0 0 0 0 1 0
s   e e e e e   m m m m m m m m m m
```

O sinal do número é obtido fazendo -1^s e neste caso o número será negativo.

O expoente é -15+19=4, portanto ao final ter-se-á que multiplicar a mantissa por $2^4 = 16$

Finalmente a mantissa, é o número 0 0 0 1 0 0 0 0 1 0, que deve ter um 1 colocado à frente dando o binário de 11 bits 1 0 0 0 1 0 0 0 0 1 0. Convertendo este número a decimal tem-se 2+64+1024=1090, que deve ser dividido por $2^{11} = 2048$ dando 1090/2048 = 0.5322265625 que multipilcado por 16 da $0.5322265625 \times 16 = 8.515625$ ou -8.515625.

Vamos ao quarto caso: 0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0 ou 0 | 1 1 0 0 0 | 0 1 1 1 0 0 1 0 1 0 O número é positivo, e seu expoente é -15+24=9. A mantissa é 1,0 1 1 1 0 0 1 0 1 0 = 1482. Ao final, o número é $+2^9 \times (1482/2048) = 370.5$.

Tente resolver agora o quinto e o sexto casos.

Como o sistema converte

Agora, a grande questão: quando você digita um número em um programa ou em uma tela, o computador precisa registrar este número na memória, usando um dos formatos aqui estudados. Vamos estudar o processo de conversão $decimal \rightarrow float$ e para campos de 16 bits. Vale a referência de este tamanho de campo não é mais usado, e só está aqui pela facilidade e simplicidade dos cálculos envolvidos. Vamos por etapas:

1. Dado um número real, por exemplo 183.27, a primeira coisa a fazer é converter este número em binário. Fica:
   $187.27_{(10)} \rightarrow 10111011.0100010100_{(2)}$

2. Agora, o ponto decimal deve ser levado à esquerda, até o primeiro 1 do número. A cada casa, o expoente deve ser incrementado. Acompanhe:
   $10111011.0100010100 \times 2^0$
   $1011101.10100010100 \times 2^1$
   $101110.110100010100 \times 2^2$
   $10111.0110100010100 \times 2^3$
   $1011.10110100010100 \times 2^4$
   $101.110110100010100 \times 2^5$
   $10.1110110100010100 \times 2^6$
   $1.01110110100010100 \times 2^7$
   $.10110110100010100 \times 2^8$
   Note que as 7 linhas acima representam o mesmo número.

3. Como o número é positivo, seu sinal é 0.

4. A potência do número é 8 que deve ser somado com 15, totalizando 23. Agora, é hora de converter o expoente em binário: $23_{(10)} \rightarrow 10111_{(2)}$

5. Finalmente, e hora de pegar 10 bits da mantissa, lembrando que o primeiro 1 (aquele que fica depois do ponto) é omitido. A mantissa fica: 0111011010

6. o numero binario em formato binary16 é
   `0 | 1 0 1 1 1 | 0 1 1 1 0 1 1 0 1 0`

7. Se você converter este número para decimal, vai encontrar 187.25, o que demonstra a baixa precisão deste formato (e explica porque ele não é usado).

☞ Para você fazer

Interprete os 2 números binários a seguir como *binary*16 inteiros e negativos e ache os valores decimais
1 e 2.

```
1 1 1 1 0 0 0 1 1 0 1 1 0 1 1 0
1 1 0 1 1 1 0 0 0 1 0 0 0 1 1 1
```

e agora interprete os 4 números a seguir como *binary*16 flutuantes e ache os decimais equivalentes com 5 casas decimais
3, 4, 5 e 6.

```
0 0 1 1 0 1 0 1 0 0 1 1 0 1 0 0
0 1 0 0 0 0 0 0 1 0 1 0 0 0 1 0
0 0 0 1 1 1 1 1 1 1 1 0 0 1 0 0
1 1 0 1 0 0 1 1 1 0 1 0 0 1 0 0
```

Finalmente, converta os valores abaixo que estão em decimal para o formato *binary*16
7 e 8.

```
103
-42.94
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.
7			
8			

CEP - UFPR - UP - UTFPR - PUCPr –
30/08/2024 - 11:15:46.2
Prof Dr P Kantek (pkantek@gmail.com)
Representação de números em máquina
VIVXq32a, V: 1.05 76575 LAURA PACHECO
DELATTRE 24CC2403 - 9 _____ / _____ /

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Agradeçamos a George Boole e a Claude Shannon por isso. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Ou seja, na matematica computacional estamos presos à matemática finita de Karl Gauss e aos grupos de Evariste Galois.

Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> dif=(((5**0.5)**2)-5)
>>> print(f'{dif:.24f}')
0.000000000000000888178420
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{lcl} 11111111 & = & -(2^6 + ... + 2^1 + 2^0) = -127 \\ \ldots & & \\ 10000001 & = & \text{-1} \\ 10000000 & = & \text{-0 (?)} \\ 00000000 & = & 0 \\ \ldots & & \\ 01111111 & = & 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e viceversa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 ... e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0$ e -0) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

Vamos ver etapa a etapa, como os resultados acima foram encontrados: Começando com o inteiro 1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1 (caso 1, acima). Como o primeiro bit é 1, o número é negativo e precisamos invertê-lo. Fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 e agora, soma-se 1 e fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1. A conversão deste número para base decimal é 1+2+8+16+128+1024+8192=9371. Portanto a resposta e -9371.

Tente achar por sua conta o segundo exemplo.

Quanto ao terceiro caso, o flutuante 1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0, devemos estudar a distribuição de bits. Vamos lá

```
1 | 1 0 0 1 1 | 0 0 0 1 0 0 0 0 1 0
s   e e e e e   m m m m m m m m m m
```

O sinal do número é obtido fazendo -1^s e neste caso o número será negativo.

O expoente é -15+19=4, portanto ao final ter-se-á que multiplicar a mantissa por $2^4 = 16$

Finalmente a mantissa, é o número 0 0 0 1 0 0 0 0 1 0, que deve ter um 1 colocado à frente dando o binário de 11 bits 1 0 0 0 1 0 0 0 0 1 0. Convertendo este número a decimal tem-se 2+64+1024=1090, que deve ser dividido por $2^{11} = 2048$ dando $1090/2048 = 0.5322265625$ que multipilcado por 16 da $0.5322265625 \times 16 = 8.515625$ ou -8.515625.

Vamos ao quarto caso: 0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0 ou `0 | 1 1 0 0 0 | 0 1 1 1 0 0 1 0 1 0` O número é positivo, e seu expoente é -15+24=9. A mantissa é 1,0 1 1 1 0 0 1 0 1 0 = 1482. Ao final, o número é $+2^9 \times (1482/2048) = 370.5$.

Tente resolver agora o quinto e o sexto casos.

Como o sistema converte

Agora, a grande questão: quando você digita um número em um programa ou em uma tela, o computador precisa registrar este número na memória, usando um dos formatos aqui estudados. Vamos estudar o processo de conversão $decimal \rightarrow float$ e para campos de 16 bits. Vale a referência de este tamanho de campo não é mais usado, e só está aqui pela facilidade e simplicidade dos cálculos envolvidos. Vamos por etapas:

1. Dado um número real, por exemplo 183.27, a primeira coisa a fazer é converter este número em binário. Fica:
   $187.27_{(10)} \rightarrow 10111011.0100010100_{(2)}$

2. Agora, o ponto decimal deve ser levado à esquerda, até o primeiro 1 do número. A cada casa, o expoente deve ser incrementado. Acompanhe:
   $10111011.0100010100 \times 2^0$
   $1011101.10100010100 \times 2^1$
   $101110.110100010100 \times 2^2$
   $10111.0110100010100 \times 2^3$
   $1011.10110100010100 \times 2^4$
   $101.110110100010100 \times 2^5$
   $10.1110110100010100 \times 2^6$
   $1.01110110100010100 \times 2^7$
   $.101110110100010100 \times 2^8$
   Note que as 7 linhas acima representam o mesmo número.

3. Como o número é positivo, seu sinal é 0.

4. A potência do número é 8 que deve ser somado com 15, totalizando 23. Agora, é hora de converter o expoente em binário: $23_{(10)} \rightarrow 10111_{(2)}$

5. Finalmente, e hora de pegar 10 bits da mantissa, lembrando que o primeiro 1 (aquele que fica depois do ponto) é omitido. A mantissa fica: 0111011010

6. o numero binario em formato binary16 é
   `0 | 1 0 1 1 1 | 0 1 1 1 0 1 1 0 1 0`

7. Se você converter este número para decimal, vai encontrar 187.25, o que demonstra a baixa precisão deste formato (e explica porque ele não é usado).

☞ Para você fazer

Interprete os 2 números binários a seguir como *binary*16 inteiros e negativos e ache os valores decimais
1 e 2.

```
1 1 1 0 0 1 0 1 1 1 1 0 1 0 0 0
1 1 1 1 1 1 1 0 0 0 1 0 0 0 1 0
```

e agora interprete os 4 números a seguir como *binary*16 flutuantes e ache os decimais equivalentes com 5 casas decimais
3, 4, 5 e 6.

```
0 0 0 1 1 0 1 0 0 1 0 0 0 1 1 1
0 0 1 0 0 1 0 0 0 1 1 0 0 1 1 0
0 0 1 1 1 0 1 0 1 0 0 1 1 1 0 0
0 0 1 1 0 0 1 1 0 0 0 0 0 0 0 1
```

Finalmente, converta os valores abaixo que estão em decimal para o formato *binary*16
7 e 8.

```
136
-40.76
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.
7			
8			

CEP - UFPR - UP - UTFPR - PUCPr –
30/08/2024 - 11:15:46.2
Prof Dr P Kantek (pkantek@gmail.com)
Representação de números em máquina
VIVXq32a, V: 1.05 76694 LAURO HENRIK
RUTZ 24CC2403 - 10 _____ / _____ / _____

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Agradeçamos a George Boole e a Claude Shannon por isso. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Ou seja, na matematica computacional estamos presos à matemática finita de Karl Gauss e aos grupos de Evariste Galois.

Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> dif=(((5**0.5)**2)-5)
>>> print(f'{dif:.24f}')
0.000000000000000888178420
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{lcl} 11111111 & = & -(2^6 + ... + 2^1 + 2^0) = -127 \\ \quad ... & & \\ 10000001 & = & \text{-1} \\ 10000000 & = & \text{-0 (?)} \\ 00000000 & = & 0 \\ \quad ... & & \\ 01111111 & = & 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e viceversa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 \cdots e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0$ e -0) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

Vamos ver etapa a etapa, como os resultados acima foram encontrados: Começando com o inteiro 1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1 (caso 1, acima). Como o primeiro bit é 1, o número é negativo e precisamos invertê-lo. Fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 e agora, soma-se 1 e fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1. A conversão deste número para base decimal é 1+2+8+16+128+1024+8192=9371. Portanto a resposta e -9371.

Tente achar por sua conta o segundo exemplo.

Quanto ao terceiro caso, o flutuante 1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0, devemos estudar a distribuição de bits. Vamos lá

```
1 | 1 0 0 1 1 | 0 0 0 1 0 0 0 0 1 0
s   e e e e e   m m m m m m m m m m
```

O sinal do número é obtido fazendo -1^s e neste caso o número será negativo.

O expoente é -15+19=4, portanto ao final ter-se-á que multiplicar a mantissa por $2^4 = 16$

Finalmente a mantissa, é o número 0 0 0 1 0 0 0 0 1 0, que deve ter um 1 colocado à frente dando o binário de 11 bits 1 0 0 0 1 0 0 0 0 1 0. Convertendo este número a decimal tem-se 2+64+1024=1090, que deve ser dividido por $2^{11} = 2048$ dando $1090/2048 = 0.5322265625$ que multipilcado por 16 da $0.5322265625 \times 16 = 8.515625$ ou -8.515625.

Vamos ao quarto caso: 0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0 ou 0 | 1 1 0 0 0 | 0 1 1 1 0 0 1 0 1 0 O número é positivo, e seu expoente é -15+24=9. A mantissa é 1,0 1 1 1 0 0 1 0 1 0 = 1482. Ao final, o número é $+2^9 \times (1482/2048) = 370.5$.

Tente resolver agora o quinto e o sexto casos.

Como o sistema converte

Agora, a grande questão: quando você digita um número em um programa ou em uma tela, o computador precisa registrar este número na memória, usando um dos formatos aqui estudados. Vamos estudar o processo de conversão $decimal \rightarrow float$ e para campos de 16 bits. Vale a referência de este tamanho de campo não é mais usado, e só está aqui pela facilidade e simplicidade dos cálculos envolvidos. Vamos por etapas:

1. Dado um número real, por exemplo 183.27, a primeira coisa a fazer é converter este número em binário. Fica:
   $187.27_{(10)} \rightarrow 10111011.0100010100_{(2)}$

2. Agora, o ponto decimal deve ser levado à esquerda, até o primeiro 1 do número. A cada casa, o expoente deve ser incrementado. Acompanhe:
   $10111011.0100010100 \times 2^0$
   $1011101.10100010100 \times 2^1$
   $101110.110100010100 \times 2^2$
   $10111.0110100010100 \times 2^3$
   $1011.10110100010100 \times 2^4$
   $101.110110100010100 \times 2^5$
   $10.1110110100010100 \times 2^6$
   $1.01110110100010100 \times 2^7$
   $.101110110100010100 \times 2^8$
   Note que as 7 linhas acima representam o mesmo número.

3. Como o número é positivo, seu sinal é 0.

4. A potência do número é 8 que deve ser somado com 15, totalizando 23. Agora, é hora de converter o expoente em binário: $23_{(10)} \rightarrow 10111_{(2)}$

5. Finalmente, e hora de pegar 10 bits da mantissa, lembrando que o primeiro 1 (aquele que fica depois do ponto) é omitido. A mantissa fica: 0111011010

6. o numero binario em formato binary16 é
   `0 | 1 0 1 1 1 | 0 1 1 1 0 1 1 0 1 0`

7. Se você converter este número para decimal, vai encontrar 187.25, o que demonstra a baixa precisão deste formato (e explica porque ele não é usado).

☞ Para você fazer

Interprete os 2 números binários a seguir como *binary*16 inteiros e negativos e ache os valores decimais
1 e 2.

```
1 1 1 0 0 1 0 0 0 1 0 1 0 0 0 0
1 1 1 0 1 1 1 0 0 1 0 0 1 0 1 1
```

e agora interprete os 4 números a seguir como *binary*16 flutuantes e ache os decimais equivalentes com 5 casas decimais
3, 4, 5 e 6.

```
1 0 0 1 0 0 1 1 0 0 0 1 1 0 1 1
1 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0
0 1 0 1 0 1 0 1 1 1 1 1 1 0 1 0 1
0 0 0 1 0 1 1 0 0 1 0 0 1 1 1 0
```

Finalmente, converta os valores abaixo que estão em decimal para o formato *binary*16
7 e 8.

```
100
-58.38
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.
7			
8			

CEP - UFPR - UP - UTFPR - PUCPr –
30/08/2024 - 11:15:46.2
Prof Dr P Kantek (pkantek@gmail.com)
Representação de números em máquina
VIVXq32a, V: 1.05 76582 LUIZ MIGUEL
OLINEK 24CC2403 - 11 _____ / _____ / _____

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Agradeçamos a George Boole e a Claude Shannon por isso. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Ou seja, na matematica computacional estamos presos à matemática finita de Karl Gauss e aos grupos de Evariste Galois.

Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> dif=(((5**0.5)**2)-5)
>>> print(f'{dif:.24f}')
0.000000000000000888178420
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{lcl} 11111111 & = & -(2^6 + ... + 2^1 + 2^0) = -127 \\ \quad ... & & \\ 10000001 & = & \text{-1} \\ 10000000 & = & \text{-0 (?)} \\ 00000000 & = & 0 \\ \quad ... & & \\ 01111111 & = & 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e viceversa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 \cdots e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0$ e -0) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

Vamos ver etapa a etapa, como os resultados acima foram encontrados: Começando com o inteiro 1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1 (caso 1, acima). Como o primeiro bit é 1, o número é negativo e precisamos invertê-lo. Fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 e agora, soma-se 1 e fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1. A conversão deste número para base decimal é 1+2+8+16+128+1024+8192=9371. Portanto a resposta e -9371.

Tente achar por sua conta o segundo exemplo.

Quanto ao terceiro caso, o flutuante 1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0, devemos estudar a distribuição de bits. Vamos lá

```
1 | 1 0 0 1 1 | 0 0 0 1 0 0 0 0 1 0
s   e e e e e   m m m m m m m m m m
```

O sinal do número é obtido fazendo -1^s e neste caso o número será negativo.

O expoente é -15+19=4, portanto ao final ter-se-á que multiplicar a mantissa por $2^4 = 16$

Finalmente a mantissa, é o número 0 0 0 1 0 0 0 0 1 0, que deve ter um 1 colocado à frente dando o binário de 11 bits 1 0 0 0 1 0 0 0 0 1 0. Convertendo este número a decimal tem-se 2+64+1024=1090, que deve ser dividido por $2^{11} = 2048$ dando $1090/2048 = 0.5322265625$ que multipilcado por 16 da $0.5322265625 \times 16 = 8.515625$ ou -8.515625.

Vamos ao quarto caso: 0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0 ou 0 | 1 1 0 0 0 | 0 1 1 1 0 0 1 0 1 0 O número é positivo, e seu expoente é -15+24=9. A mantissa é 1,0 1 1 1 0 0 1 0 1 0 = 1482. Ao final, o número é $+2^9 \times (1482/2048) = 370.5$.

Tente resolver agora o quinto e o sexto casos.

Como o sistema converte

Agora, a grande questão: quando você digita um número em um programa ou em uma tela, o computador precisa registrar este número na memória, usando um dos formatos aqui estudados. Vamos estudar o processo de conversão $decimal \rightarrow float$ e para campos de 16 bits. Vale a referência de este tamanho de campo não é mais usado, e só está aqui pela facilidade e simplicidade dos cálculos envolvidos. Vamos por etapas:

1. Dado um número real, por exemplo 183.27, a primeira coisa a fazer é converter este número em binário. Fica:
   $187.27_{(10)} \rightarrow 10111011.0100010100_{(2)}$

2. Agora, o ponto decimal deve ser levado à esquerda, até o primeiro 1 do número. A cada casa, o expoente deve ser incrementado. Acompanhe:
   $10111011.0100010100 \times 2^0$
   $1011101.10100010100 \times 2^1$
   $101110.110100010100 \times 2^2$
   $10111.0110100010100 \times 2^3$
   $1011.10110100010100 \times 2^4$
   $101.110110100010100 \times 2^5$
   $10.1110110100010100 \times 2^6$
   $1.01110110100010100 \times 2^7$
   $.101110110100010100 \times 2^8$
   Note que as 7 linhas acima representam o mesmo número.

3. Como o número é positivo, seu sinal é 0.

4. A potência do número é 8 que deve ser somado com 15, totalizando 23. Agora, é hora de converter o expoente em binário: $23_{(10)} \rightarrow 10111_{(2)}$

5. Finalmente, e hora de pegar 10 bits da mantissa, lembrando que o primeiro 1 (aquele que fica depois do ponto) é omitido. A mantissa fica: 0111011010

6. o numero binario em formato binary16 é
   `0 | 1 0 1 1 1 | 0 1 1 1 0 1 1 0 1 0`

7. Se você converter este número para decimal, vai encontrar 187.25, o que demonstra a baixa precisão deste formato (e explica porque ele não é usado).

☞ Para você fazer

Interprete os 2 números binários a seguir como $binary16$ inteiros e negativos e ache os valores decimais
1 e 2.

```
1 1 0 1 1 1 1 1 1 1 1 1 0 0 1 0 0
1 1 1 0 0 1 1 0 1 0 0 1 0 0 1 1
```

e agora interprete os 4 números a seguir como $binary16$ flutuantes e ache os decimais equivalentes com 5 casas decimais
3, 4, 5 e 6.

```
0 1 0 1 0 1 0 1 0 1 1 0 0 0 0 1
0 0 1 0 0 0 0 0 0 1 0 0 1 0 1 1
0 0 1 1 0 0 1 0 1 1 1 0 1 1 0 1
1 0 0 0 0 0 0 0 1 1 1 1 0 0 1 0
```

Finalmente, converta os valores abaixo que estão em decimal para o formato $binary16$
7 e 8.

```
106
-49.76
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.
7			
8			

CEP - UFPR - UP - UTFPR - PUCPr –
30/08/2024 - 11:15:46.2
Prof Dr P Kantek (pkantek@gmail.com)
Representação de números em máquina
VIVXq32a, V: 1.05 76599 NICOLE DA SILVA DE AZEVEDO 24CC2403 - 12 _____ / _____ /

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Agradeçamos a George Boole e a Claude Shannon por isso. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Ou seja, na matematica computacional estamos presos à matemática finita de Karl Gauss e aos grupos de Evariste Galois.

Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> dif=(((5**0.5)**2)-5)
>>> print(f'{dif:.24f}')
0.000000000000000888178420
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{lcl} 11111111 & = & -(2^6 + ... + 2^1 + 2^0) = -127 \\ \ldots & & \\ 10000001 & = & \text{-1} \\ 10000000 & = & \text{-0 (?)} \\ 00000000 & = & 0 \\ \ldots & & \\ 01111111 & = & 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e viceversa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 ... e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0$ e -0) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

Vamos ver etapa a etapa, como os resultados acima foram encontrados: Começando com o inteiro 1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1 (caso 1, acima). Como o primeiro bit é 1, o número é negativo e precisamos invertê-lo. Fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 e agora, soma-se 1 e fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1. A conversão deste número para base decimal é 1+2+8+16+128+1024+8192=9371. Portanto a resposta e -9371.

Tente achar por sua conta o segundo exemplo.

Quanto ao terceiro caso, o flutuante 1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0, devemos estudar a distribuição de bits. Vamos lá

```
1 | 1 0 0 1 1 | 0 0 0 1 0 0 0 0 1 0
s   e e e e e   m m m m m m m m m m
```

O sinal do número é obtido fazendo -1^s e neste caso o número será negativo.

O expoente é -15+19=4, portanto ao final ter-se-á que multiplicar a mantissa por $2^4 = 16$

Finalmente a mantissa, é o número 0 0 0 1 0 0 0 0 1 0, que deve ter um 1 colocado à frente dando o binário de 11 bits 1 0 0 0 1 0 0 0 0 1 0. Convertendo este número a decimal tem-se 2+64+1024=1090, que deve ser dividido por $2^{11} = 2048$ dando $1090/2048 = 0.5322265625$ que multipilcado por 16 da $0.5322265625 \times 16 = 8.515625$ ou -8.515625.

Vamos ao quarto caso: 0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0 ou `0 | 1 1 0 0 0 | 0 1 1 1 0 0 1 0 1 0` O número é positivo, e seu expoente é -15+24=9. A mantissa é 1,0 1 1 1 0 0 1 0 1 0 = 1482. Ao final, o número é $+2^9 \times (1482/2048) = 370.5$.

Tente resolver agora o quinto e o sexto casos.

Como o sistema converte

Agora, a grande questão: quando você digita um número em um programa ou em uma tela, o computador precisa registrar este número na memória, usando um dos formatos aqui estudados. Vamos estudar o processo de conversão $decimal \rightarrow float$ e para campos de 16 bits. Vale a referência de este tamanho de campo não é mais usado, e só está aqui pela facilidade e simplicidade dos cálculos envolvidos. Vamos por etapas:

1. Dado um número real, por exemplo 183.27, a primeira coisa a fazer é converter este número em binário. Fica:
   $187.27_{(10)} \rightarrow 10111011.0100010100_{(2)}$

2. Agora, o ponto decimal deve ser levado à esquerda, até o primeiro 1 do número. A cada casa, o expoente deve ser incrementado. Acompanhe:
   $10111011.0100010100 \times 2^0$
   $1011101.10100010100 \times 2^1$
   $101110.110100010100 \times 2^2$
   $10111.0110100010100 \times 2^3$
   $1011.10110100010100 \times 2^4$
   $101.110110100010100 \times 2^5$
   $10.1110110100010100 \times 2^6$
   $1.01110110100010100 \times 2^7$
   $.101110110100010100 \times 2^8$
   Note que as 7 linhas acima representam o mesmo número.

3. Como o número é positivo, seu sinal é 0.

4. A potência do número é 8 que deve ser somado com 15, totalizando 23. Agora, é hora de converter o expoente em binário: $23_{(10)} \rightarrow 10111_{(2)}$

5. Finalmente, e hora de pegar 10 bits da mantissa, lembrando que o primeiro 1 (aquele que fica depois do ponto) é omitido. A mantissa fica: 0111011010

6. o numero binario em formato binary16 é
   `0 | 1 0 1 1 1 | 0 1 1 1 0 1 1 0 1 0`

7. Se você converter este número para decimal, vai encontrar 187.25, o que demonstra a baixa precisão deste formato (e explica porque ele não é usado).

☞ Para você fazer

Interprete os 2 números binários a seguir como $binary16$ inteiros e negativos e ache os valores decimais
1 e 2.

```
1 1 0 1 1 0 1 0 0 1 0 1 1 1 0 1
1 1 1 0 1 1 1 0 1 1 0 1 0 0 0 1
```

e agora interprete os 4 números a seguir como $binary16$ flutuantes e ache os decimais equivalentes com 5 casas decimais
3, 4, 5 e 6.

```
0 0 0 0 1 1 0 0 1 0 1 0 0 1 0 1
1 0 1 0 1 0 1 1 1 1 1 0 0 1 1 0
1 0 0 0 0 1 0 0 1 0 1 0 1 1 0 0
0 0 0 0 0 0 1 1 1 1 0 1 0 1 0 0
```

Finalmente, converta os valores abaixo que estão em decimal para o formato $binary16$
7 e 8.

```
120
-48.74
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.
7			
8			

CEP - UFPR - UP - UTFPR - PUCPr –
30/08/2024 - 11:15:46.2
Prof Dr P Kantek (pkantek@gmail.com)
Representação de números em máquina
VIVXq32a, V: 1.05 76601 PAULO JOSE FREIRE CORREA 24CC2403 - 13 _____ / _____ /

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Agradeçamos a George Boole e a Claude Shannon por isso. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Ou seja, na matematica computacional estamos presos à matemática finita de Karl Gauss e aos grupos de Evariste Galois.

Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> dif=(((5**0.5)**2)-5)
>>> print(f'{dif:.24f}')
0.000000000000000888178420
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{lcl} 11111111 & = & -(2^6 + ... + 2^1 + 2^0) = -127 \\ \ldots & & \\ 10000001 & = & \text{-1} \\ 10000000 & = & \text{-0 (?)} \\ 00000000 & = & 0 \\ \ldots & & \\ 01111111 & = & 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e viceversa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 ... e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0$ e -0) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

Vamos ver etapa a etapa, como os resultados acima foram encontrados: Começando com o inteiro 1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1 (caso 1, acima). Como o primeiro bit é 1, o número é negativo e precisamos invertê-lo. Fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 e agora, soma-se 1 e fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1. A conversão deste número para base decimal é 1+2+8+16+128+1024+8192=9371. Portanto a resposta e -9371.

Tente achar por sua conta o segundo exemplo.

Quanto ao terceiro caso, o flutuante 1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0, devemos estudar a distribuição de bits. Vamos lá

```
1 | 1 0 0 1 1 | 0 0 0 1 0 0 0 0 1 0
s   e e e e e   m m m m m m m m m m
```

O sinal do número é obtido fazendo -1^s e neste caso o número será negativo.

O expoente é -15+19=4, portanto ao final ter-se-á que multiplicar a mantissa por $2^4 = 16$

Finalmente a mantissa, é o número 0 0 0 1 0 0 0 0 1 0, que deve ter um 1 colocado à frente dando o binário de 11 bits 1 0 0 0 1 0 0 0 0 1 0. Convertendo este número a decimal tem-se 2+64+1024=1090, que deve ser dividido por $2^{11} = 2048$ dando $1090/2048 = 0.5322265625$ que multipilcado por 16 da $0.5322265625 \times 16 = 8.515625$ ou -8.515625.

Vamos ao quarto caso: 0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0 ou `0 | 1 1 0 0 0 | 0 1 1 1 0 0 1 0 1 0` O número é positivo, e seu expoente é -15+24=9. A mantissa é 1,0 1 1 1 0 0 1 0 1 0 = 1482. Ao final, o número é $+2^9 \times (1482/2048) = 370.5$.

Tente resolver agora o quinto e o sexto casos.

Como o sistema converte

Agora, a grande questão: quando você digita um número em um programa ou em uma tela, o computador precisa registrar este número na memória, usando um dos formatos aqui estudados. Vamos estudar o processo de conversão $decimal \rightarrow float$ e para campos de 16 bits. Vale a referência de este tamanho de campo não é mais usado, e só está aqui pela facilidade e simplicidade dos cálculos envolvidos. Vamos por etapas:

1. Dado um número real, por exemplo 183.27, a primeira coisa a fazer é converter este número em binário. Fica:
   $187.27_{(10)} \rightarrow 10111011.0100010100_{(2)}$

2. Agora, o ponto decimal deve ser levado à esquerda, até o primeiro 1 do número. A cada casa, o expoente deve ser incrementado. Acompanhe:
   $10111011.0100010100 \times 2^0$
   $1011101.10100010100 \times 2^1$
   $101110.110100010100 \times 2^2$
   $10111.0110100010100 \times 2^3$
   $1011.10110100010100 \times 2^4$
   $101.110110100010100 \times 2^5$
   $10.1110110100010100 \times 2^6$
   $1.01110110100010100 \times 2^7$
   $.101110110100010100 \times 2^8$
   Note que as 7 linhas acima representam o mesmo número.

3. Como o número é positivo, seu sinal é 0.

4. A potência do número é 8 que deve ser somado com 15, totalizando 23. Agora, é hora de converter o expoente em binário: $23_{(10)} \rightarrow 10111_{(2)}$

5. Finalmente, e hora de pegar 10 bits da mantissa, lembrando que o primeiro 1 (aquele que fica depois do ponto) é omitido. A mantissa fica: 0111011010

6. o numero binario em formato binary16 é
   `0 | 1 0 1 1 1 | 0 1 1 1 0 1 1 0 1 0`

7. Se você converter este número para decimal, vai encontrar 187.25, o que demonstra a baixa precisão deste formato (e explica porque ele não é usado).

☞ Para você fazer

Interprete os 2 números binários a seguir como $binary16$ inteiros e negativos e ache os valores decimais
1 e 2.

```
1 1 0 1 1 1 1 0 1 1 1 1 1 1 0 1
1 1 1 1 0 0 1 1 0 1 0 1 1 1 0 0
```

e agora interprete os 4 números a seguir como $binary16$ flutuantes e ache os decimais equivalentes com 5 casas decimais
3, 4, 5 e 6.

```
1 1 0 1 0 0 0 0 1 1 0 1 0 0 1 0
0 0 1 1 1 1 0 0 1 1 1 0 0 1 0 0 0
0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 1 1
0 1 0 0 1 0 1 0 0 0 0 1 0 0 1 1
```

Finalmente, converta os valores abaixo que estão em decimal para o formato $binary16$
7 e 8.

```
114
-57.8
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.
7			
8			

CEP - UFPR - UP - UTFPR - PUCPr – 30/08/2024 - 11:15:46.2
Prof Dr P Kantek (pkantek@gmail.com)
Representação de números em máquina
VIVXq32a, V: 1.05 76618 PEDRO HENRIQUE PIEKARSKI DE OL 24CC2403 - 14 _____ / / _____

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Agradeçamos a George Boole e a Claude Shannon por isso. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Ou seja, na matematica computacional estamos presos à matemática finita de Karl Gauss e aos grupos de Evariste Galois.

Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> dif=(((5**0.5)**2)-5)
>>> print(f'{dif:.24f}')
0.000000000000000888178420
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{lcl} 11111111 & = & -(2^6 + ... + 2^1 + 2^0) = -127 \\ \ldots & & \\ 10000001 & = & \text{-1} \\ 10000000 & = & \text{-0 (?)} \\ 00000000 & = & 0 \\ \ldots & & \\ 01111111 & = & 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e viceversa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 ... e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0$ e -0) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

Vamos ver etapa a etapa, como os resultados acima foram encontrados: Começando com o inteiro 1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1 (caso 1, acima). Como o primeiro bit é 1, o número é negativo e precisamos invertê-lo. Fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 e agora, soma-se 1 e fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1. A conversão deste número para base decimal é 1+2+8+16+128+1024+8192=9371. Portanto a resposta e -9371.

Tente achar por sua conta o segundo exemplo.

Quanto ao terceiro caso, o flutuante 1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0, devemos estudar a distribuição de bits. Vamos lá

```
1 | 1 0 0 1 1 | 0 0 0 1 0 0 0 0 1 0
s   e e e e e   m m m m m m m m m m
```

O sinal do número é obtido fazendo -1^s e neste caso o número será negativo.

O expoente é -15+19=4, portanto ao final ter-se-á que multiplicar a mantissa por $2^4 = 16$

Finalmente a mantissa, é o número 0 0 0 1 0 0 0 0 1 0, que deve ter um 1 colocado à frente dando o binário de 11 bits 1 0 0 0 1 0 0 0 0 1 0. Convertendo este número a decimal tem-se 2+64+1024=1090, que deve ser dividido por $2^{11} = 2048$ dando $1090/2048 = 0.5322265625$ que multipilcado por 16 da $0.5322265625 \times 16 = 8.515625$ ou -8.515625.

Vamos ao quarto caso: 0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0 ou `0 | 1 1 0 0 0 | 0 1 1 1 0 0 1 0 1 0` O número é positivo, e seu expoente é -15+24=9. A mantissa é 1,0 1 1 1 0 0 1 0 1 0 = 1482. Ao final, o número é $+2^9 \times (1482/2048) = 370.5$.

Tente resolver agora o quinto e o sexto casos.

Como o sistema converte

Agora, a grande questão: quando você digita um número em um programa ou em uma tela, o computador precisa registrar este número na memória, usando um dos formatos aqui estudados. Vamos estudar o processo de conversão $decimal \rightarrow float$ e para campos de 16 bits. Vale a referência de este tamanho de campo não é mais usado, e só está aqui pela facilidade e simplicidade dos cálculos envolvidos. Vamos por etapas:

1. Dado um número real, por exemplo 183.27, a primeira coisa a fazer é converter este número em binário. Fica:
   $187.27_{(10)} \rightarrow 10111011.0100010100_{(2)}$

2. Agora, o ponto decimal deve ser levado à esquerda, até o primeiro 1 do número. A cada casa, o expoente deve ser incrementado. Acompanhe:
   $10111011.0100010100 \times 2^0$
   $1011101.10100010100 \times 2^1$
   $101110.110100010100 \times 2^2$
   $10111.0110100010100 \times 2^3$
   $1011.10110100010100 \times 2^4$
   $101.110110100010100 \times 2^5$
   $10.1110110100010100 \times 2^6$
   $1.01110110100010100 \times 2^7$
   $.101110110100010100 \times 2^8$
   Note que as 7 linhas acima representam o mesmo número.

3. Como o número é positivo, seu sinal é 0.

4. A potência do número é 8 que deve ser somado com 15, totalizando 23. Agora, é hora de converter o expoente em binário: $23_{(10)} \rightarrow 10111_{(2)}$

5. Finalmente, e hora de pegar 10 bits da mantissa, lembrando que o primeiro 1 (aquele que fica depois do ponto) é omitido. A mantissa fica: 0111011010

6. o numero binario em formato binary16 é
   `0 | 1 0 1 1 1 | 0 1 1 1 0 1 1 0 1 0`

7. Se você converter este número para decimal, vai encontrar 187.25, o que demonstra a baixa precisão deste formato (e explica porque ele não é usado).

☞ Para você fazer

Interprete os 2 números binários a seguir como *binary*16 inteiros e negativos e ache os valores decimais
1 e 2.

```
1 1 1 1 0 0 0 1 0 0 1 0 1 0 0 1
1 1 0 1 1 1 1 0 1 1 1 0 0 0 1 0
```

e agora interprete os 4 números a seguir como *binary*16 flutuantes e ache os decimais equivalentes com 5 casas decimais
3, 4, 5 e 6.

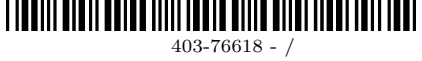
```
0 1 0 0 1 1 0 1 0 1 1 1 0 1 1 0
0 1 0 0 0 1 1 0 0 1 1 0 1 0 1 1
1 1 0 1 1 1 1 1 0 0 1 1 0 1 0 0
1 0 1 1 0 1 0 0 0 0 0 0 1 0 1 0
```

Finalmente, converta os valores abaixo que estão em decimal para o formato *binary*16
7 e 8.

```
116
-57.3
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.
7			
8			



CEP - UFPR - UP - UTFPR - PUCPr –
30/08/2024 - 11:15:46.2
Prof Dr P Kantek (pkantek@gmail.com)
Representação de números em máquina
VIVXq32a, V: 1.05 76625 RAUL KNAPKI
CUNHA 24CC2403 - 15 _____ / _____ / _____

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Agradeçamos a George Boole e a Claude Shannon por isso. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Ou seja, na matematica computacional estamos presos à matemática finita de Karl Gauss e aos grupos de Evariste Galois.

Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> dif=(((5**0.5)**2)-5)
>>> print(f'{dif:.24f}')
0.000000000000000888178420
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{lcl} 11111111 & = & -(2^6 + ... + 2^1 + 2^0) = -127 \\ \quad ... & & \\ 10000001 & = & \text{-1} \\ 10000000 & = & \text{-0 (?)} \\ 00000000 & = & 0 \\ \quad ... & & \\ 01111111 & = & 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e viceversa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 \cdots e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0$ e -0) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

Vamos ver etapa a etapa, como os resultados acima foram encontrados: Começando com o inteiro 1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1 (caso 1, acima). Como o primeiro bit é 1, o número é negativo e precisamos invertê-lo. Fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 e agora, soma-se 1 e fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1. A conversão deste número para base decimal é 1+2+8+16+128+1024+8192=9371. Portanto a resposta e -9371.

Tente achar por sua conta o segundo exemplo.

Quanto ao terceiro caso, o flutuante 1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0, devemos estudar a distribuição de bits. Vamos lá

```
1 | 1 0 0 1 1 | 0 0 0 1 0 0 0 0 1 0
s   e e e e e   m m m m m m m m m m
```

O sinal do número é obtido fazendo -1^s e neste caso o número será negativo.

O expoente é -15+19=4, portanto ao final ter-se-á que multiplicar a mantissa por $2^4 = 16$

Finalmente a mantissa, é o número 0 0 0 1 0 0 0 0 1 0, que deve ter um 1 colocado à frente dando o binário de 11 bits 1 0 0 0 1 0 0 0 0 1 0. Convertendo este número a decimal tem-se 2+64+1024=1090, que deve ser dividido por $2^{11} = 2048$ dando 1090/2048 = 0.5322265625 que multipilcado por 16 da $0.5322265625 \times 16 = 8.515625$ ou -8.515625.

Vamos ao quarto caso: 0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0 ou 0 | 1 1 0 0 0 | 0 1 1 1 0 0 1 0 1 0 O número é positivo, e seu expoente é -15+24=9. A mantissa é 1,0 1 1 1 0 0 1 0 1 0 = 1482. Ao final, o número é $+2^9 \times (1482/2048) = 370.5$.

Tente resolver agora o quinto e o sexto casos.

Como o sistema converte

Agora, a grande questão: quando você digita um número em um programa ou em uma tela, o computador precisa registrar este número na memória, usando um dos formatos aqui estudados. Vamos estudar o processo de conversão $decimal \rightarrow float$ e para campos de 16 bits. Vale a referência de este tamanho de campo não é mais usado, e só está aqui pela facilidade e simplicidade dos cálculos envolvidos. Vamos por etapas:

1. Dado um número real, por exemplo 183.27, a primeira coisa a fazer é converter este número em binário. Fica:
   $187.27_{(10)} \rightarrow 10111011.0100010100_{(2)}$

2. Agora, o ponto decimal deve ser levado à esquerda, até o primeiro 1 do número. A cada casa, o expoente deve ser incrementado. Acompanhe:
   $10111011.0100010100 \times 2^0$
   $1011101.10100010100 \times 2^1$
   $101110.110100010100 \times 2^2$
   $10111.0110100010100 \times 2^3$
   $1011.10110100010100 \times 2^4$
   $101.110110100010100 \times 2^5$
   $10.1110110100010100 \times 2^6$
   $1.01110110100010100 \times 2^7$
   $.101110110100010100 \times 2^8$
   Note que as 7 linhas acima representam o mesmo número.

3. Como o número é positivo, seu sinal é 0.

4. A potência do número é 8 que deve ser somado com 15, totalizando 23. Agora, é hora de converter o expoente em binário: $23_{(10)} \rightarrow 10111_{(2)}$

5. Finalmente, e hora de pegar 10 bits da mantissa, lembrando que o primeiro 1 (aquele que fica depois do ponto) é omitido. A mantissa fica: 0111011010

6. o numero binario em formato binary16 é
   `0 | 1 0 1 1 1 | 0 1 1 1 0 1 1 0 1 0`

7. Se você converter este número para decimal, vai encontrar 187.25, o que demonstra a baixa precisão deste formato (e explica porque ele não é usado).

☞ Para você fazer

Interprete os 2 números binários a seguir como *binary*16 inteiros e negativos e ache os valores decimais
1 e 2.

```
1 1 1 0 0 0 0 0 1 0 1 1 0 0 0 1
1 1 0 1 1 1 0 1 1 1 1 0 0 1 0 0
```

e agora interprete os 4 números a seguir como *binary*16 flutuantes e ache os decimais equivalentes com 5 casas decimais
3, 4, 5 e 6.

```
1 1 0 0 1 1 1 0 0 1 0 0 1 0 0 1
0 0 1 1 0 0 0 0 1 1 1 0 1 1 0 1
0 0 0 1 1 1 1 1 1 0 0 0 0 1 0 0
1 0 1 0 1 0 1 1 0 1 0 0 0 1 0 1
```

Finalmente, converta os valores abaixo que estão em decimal para o formato *binary*16
7 e 8.

```
137
-59.56
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.
7			
8			

CEP - UFPR - UP - UTFPR - PUCPr –
30/08/2024 - 11:15:46.2
Prof Dr P Kantek (pkantek@gmail.com)
Representação de números em máquina
VIVXq32a, V: 1.05 76632 VICTOR BRONDANI DOS SANTOS 24CC2403 - 16 _____ / _____ /

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Agradeçamos a George Boole e a Claude Shannon por isso. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Ou seja, na matematica computacional estamos presos à matemática finita de Karl Gauss e aos grupos de Evariste Galois.

Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> dif=(((5**0.5)**2)-5)
>>> print(f'{dif:.24f}')
0.000000000000000888178420
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{lcl} 11111111 & = & -(2^6 + ... + 2^1 + 2^0) = -127 \\ \ldots & & \\ 10000001 & = & \text{-1} \\ 10000000 & = & \text{-0 (?)} \\ 00000000 & = & 0 \\ \ldots & & \\ 01111111 & = & 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e viceversa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 ... e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0$ e -0) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

Vamos ver etapa a etapa, como os resultados acima foram encontrados: Começando com o inteiro 1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1 (caso 1, acima). Como o primeiro bit é 1, o número é negativo e precisamos invertê-lo. Fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 e agora, soma-se 1 e fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1. A conversão deste número para base decimal é 1+2+8+16+128+1024+8192=9371. Portanto a resposta e -9371.

Tente achar por sua conta o segundo exemplo.

Quanto ao terceiro caso, o flutuante 1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0, devemos estudar a distribuição de bits. Vamos lá

```
1 | 1 0 0 1 1 | 0 0 0 1 0 0 0 0 1 0
s   e e e e e   m m m m m m m m m m
```

O sinal do número é obtido fazendo -1^s e neste caso o número será negativo.

O expoente é -15+19=4, portanto ao final ter-se-á que multiplicar a mantissa por $2^4 = 16$

Finalmente a mantissa, é o número 0 0 0 1 0 0 0 0 1 0, que deve ter um 1 colocado à frente dando o binário de 11 bits 1 0 0 0 1 0 0 0 0 1 0. Convertendo este número a decimal tem-se 2+64+1024=1090, que deve ser dividido por $2^{11} = 2048$ dando $1090/2048 = 0.5322265625$ que multipilcado por 16 da $0.5322265625 \times 16 = 8.515625$ ou -8.515625.

Vamos ao quarto caso: 0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0 ou `0 | 1 1 0 0 0 | 0 1 1 1 0 0 1 0 1 0` O número é positivo, e seu expoente é -15+24=9. A mantissa é 1,0 1 1 1 0 0 1 0 1 0 = 1482. Ao final, o número é $+2^9 \times (1482/2048) = 370.5$.

Tente resolver agora o quinto e o sexto casos.

Como o sistema converte

Agora, a grande questão: quando você digita um número em um programa ou em uma tela, o computador precisa registrar este número na memória, usando um dos formatos aqui estudados. Vamos estudar o processo de conversão $decimal \rightarrow float$ e para campos de 16 bits. Vale a referência de este tamanho de campo não é mais usado, e só está aqui pela facilidade e simplicidade dos cálculos envolvidos. Vamos por etapas:

1. Dado um número real, por exemplo 183.27, a primeira coisa a fazer é converter este número em binário. Fica:
   $187.27_{(10)} \rightarrow 10111011.0100010100_{(2)}$

2. Agora, o ponto decimal deve ser levado à esquerda, até o primeiro 1 do número. A cada casa, o expoente deve ser incrementado. Acompanhe:
   $10111011.0100010100 \times 2^0$
   $1011101.10100010100 \times 2^1$
   $101110.110100010100 \times 2^2$
   $10111.0110100010100 \times 2^3$
   $1011.10110100010100 \times 2^4$
   $101.110110100010100 \times 2^5$
   $10.1110110100010100 \times 2^6$
   $1.01110110100010100 \times 2^7$
   $.101110110100010100 \times 2^8$
   Note que as 7 linhas acima representam o mesmo número.

3. Como o número é positivo, seu sinal é 0.

4. A potência do número é 8 que deve ser somado com 15, totalizando 23. Agora, é hora de converter o expoente em binário: $23_{(10)} \rightarrow 10111_{(2)}$

5. Finalmente, e hora de pegar 10 bits da mantissa, lembrando que o primeiro 1 (aquele que fica depois do ponto) é omitido. A mantissa fica: 0111011010

6. o numero binario em formato binary16 é
   `0 | 1 0 1 1 1 | 0 1 1 1 0 1 1 0 1 0`

7. Se você converter este número para decimal, vai encontrar 187.25, o que demonstra a baixa precisão deste formato (e explica porque ele não é usado).

☞ Para você fazer

Interprete os 2 números binários a seguir como $binary16$ inteiros e negativos e ache os valores decimais
1 e 2.

```
1 1 1 0 1 1 1 1 0 1 0 0 1 1 1 0
1 1 1 1 0 0 0 1 1 0 0 0 0 1 1 1
```

e agora interprete os 4 números a seguir como $binary16$ flutuantes e ache os decimais equivalentes com 5 casas decimais
3, 4, 5 e 6.

```
0 1 0 0 0 1 1 0 0 0 1 1 1 1 1 0
1 0 0 0 1 1 1 0 0 0 0 0 1 0 0 0
0 1 1 0 1 0 1 0 1 1 0 1 0 1 1 1
1 0 0 1 1 1 0 1 1 0 1 0 1 1 1 0
```

Finalmente, converta os valores abaixo que estão em decimal para o formato $binary16$
7 e 8.

```
97
-57.84
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.
7			
8			

CEP - UFPR - UP - UTFPR - PUCPr – 30/08/2024 - 11:15:46.2
Prof Dr P Kantek (pkantek@gmail.com)
Representação de números em máquina
VIVXq32a, V: 1.05 76649 VICTOR HUGO DOS SANTOS DE CAMA 24CC2403 - 17 _____ / / _____

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Agradeçamos a George Boole e a Claude Shannon por isso. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Ou seja, na matematica computacional estamos presos à matemática finita de Karl Gauss e aos grupos de Evariste Galois.

Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> dif=(((5**0.5)**2)-5)
>>> print(f'{dif:.24f}')
0.000000000000000888178420
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{lcl} 11111111 & = & -(2^6 + ... + 2^1 + 2^0) = -127 \\ \quad ... & & \\ 10000001 & = & \text{-1} \\ 10000000 & = & \text{-0 (?)} \\ 00000000 & = & 0 \\ \quad ... & & \\ 01111111 & = & 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e viceversa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 ... e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0$ e -0) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

Vamos ver etapa a etapa, como os resultados acima foram encontrados: Começando com o inteiro 1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1 (caso 1, acima). Como o primeiro bit é 1, o número é negativo e precisamos invertê-lo. Fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 e agora, soma-se 1 e fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1. A conversão deste número para base decimal é 1+2+8+16+128+1024+8192=9371. Portanto a resposta e -9371.

Tente achar por sua conta o segundo exemplo.

Quanto ao terceiro caso, o flutuante 1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0, devemos estudar a distribuição de bits. Vamos lá

```
1 | 1 0 0 1 1 | 0 0 0 1 0 0 0 0 1 0
s   e e e e e   m m m m m m m m m m
```

O sinal do número é obtido fazendo -1^s e neste caso o número será negativo.

O expoente é -15+19=4, portanto ao final ter-se-á que multiplicar a mantissa por $2^4 = 16$

Finalmente a mantissa, é o número 0 0 0 1 0 0 0 0 1 0, que deve ter um 1 colocado à frente dando o binário de 11 bits 1 0 0 0 1 0 0 0 0 1 0. Convertendo este número a decimal tem-se 2+64+1024=1090, que deve ser dividido por $2^{11} = 2048$ dando $1090/2048 = 0.5322265625$ que multipilcado por 16 da $0.5322265625 \times 16 = 8.515625$ ou -8.515625.

Vamos ao quarto caso: 0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0 ou `0 | 1 1 0 0 0 | 0 1 1 1 0 0 1 0 1 0` O número é positivo, e seu expoente é -15+24=9. A mantissa é 1,0 1 1 1 0 0 1 0 1 0 = 1482. Ao final, o número é $+2^9 \times (1482/2048) = 370.5$.

Tente resolver agora o quinto e o sexto casos.

Como o sistema converte

Agora, a grande questão: quando você digita um número em um programa ou em uma tela, o computador precisa registrar este número na memória, usando um dos formatos aqui estudados. Vamos estudar o processo de conversão $decimal \rightarrow float$ e para campos de 16 bits. Vale a referência de este tamanho de campo não é mais usado, e só está aqui pela facilidade e simplicidade dos cálculos envolvidos. Vamos por etapas:

1. Dado um número real, por exemplo 183.27, a primeira coisa a fazer é converter este número em binário. Fica:
   $187.27_{(10)} \rightarrow 10111011.0100010100_{(2)}$

2. Agora, o ponto decimal deve ser levado à esquerda, até o primeiro 1 do número. A cada casa, o expoente deve ser incrementado. Acompanhe:
   $10111011.0100010100 \times 2^0$
   $1011101.10100010100 \times 2^1$
   $101110.110100010100 \times 2^2$
   $10111.0110100010100 \times 2^3$
   $1011.10110100010100 \times 2^4$
   $101.110110100010100 \times 2^5$
   $10.1110110100010100 \times 2^6$
   $1.01110110100010100 \times 2^7$
   $.101110110100010100 \times 2^8$
   Note que as 7 linhas acima representam o mesmo número.

3. Como o número é positivo, seu sinal é 0.

4. A potência do número é 8 que deve ser somado com 15, totalizando 23. Agora, é hora de converter o expoente em binário: $23_{(10)} \rightarrow 10111_{(2)}$

5. Finalmente, e hora de pegar 10 bits da mantissa, lembrando que o primeiro 1 (aquele que fica depois do ponto) é omitido. A mantissa fica: 0111011010

6. o numero binario em formato binary16 é
   `0 | 1 0 1 1 1 | 0 1 1 1 0 1 1 0 1 0`

7. Se você converter este número para decimal, vai encontrar 187.25, o que demonstra a baixa precisão deste formato (e explica porque ele não é usado).

☞ Para você fazer

Interprete os 2 números binários a seguir como $binary16$ inteiros e negativos e ache os valores decimais
1 e 2.

```
1 1 1 0 1 0 1 0 0 0 1 0 0 1 1 1
1 1 1 1 0 0 1 0 0 1 1 0 1 0 0 1
```

e agora interprete os 4 números a seguir como $binary16$ flutuantes e ache os decimais equivalentes com 5 casas decimais
3, 4, 5 e 6.

```
1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 0
1 0 0 0 0 0 1 1 0 1 1 1 1 1 1 1
1 0 0 1 1 1 1 0 0 0 1 1 1 0 1 1
0 1 0 0 0 1 0 0 1 0 0 1 1 1 0 1
```

Finalmente, converta os valores abaixo que estão em decimal para o formato $binary16$
7 e 8.

```
107
-44.22
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.
7			
8			

CEP - UFPR - UP - UTFPR - PUCPr –
30/08/2024 - 11:15:46.2
Prof Dr P Kantek (pkantek@gmail.com)
Representação de números em máquina
VIVXq32a, V: 1.05 76656 VITOR EDUARDO DE GOES FONTES 24CC2403 - 18 _____ / _____ /

Representação de números em máquinas

Qualquer máquina digital (computadores, celulares, fornos de microondas, impressoras, I-pads, automóveis, aviões, máquinas fotográficas, geladeiras, televisores, máquinas de dinheiro,...) precisa manusear e armazenar números. Embora lá nos primórdios do século XX tenha havido a tentação de representar números por medidas analógicas, rapidamente a qualidade da representação digital se impôs.

Todas essas máquinas usam lógica binária e portanto usam bits (0 ou 1) internamente. Agradeçamos a George Boole e a Claude Shannon por isso. Em geral, para guardar um número, todas trabalham com quantidades fixas de bits, o que ao fim e ao cabo implica em uma quantidade finita de números representáveis. Ou seja, na matematica computacional estamos presos à matemática finita de Karl Gauss e aos grupos de Evariste Galois.

Todos os demais números são obtidos por proximidade a algum número desses que é representável, gerando aqui um erro de representação. Por exemplo, 2, seu quadrado e o número 4 podem ser representados em um computador usando Python. Já, a raiz quadrada de 5 não pode (e quando obtida é forçosamente arredondada) daí:

```
>>> 2**2 == 4
True
>>> ((5**0.5)**2) == 5
False
>>> dif=(((5**0.5)**2)-5)
>>> print(f'{dif:.24f}')
0.000000000000000888178420
```

Inteiros

Os inteiros podem ser corretamente representados, limitados ao tamanho do **registro** assim entendido como a unidade de bits que são alocados ao armazenamento de um número. Supondo n bits, pode-se guardar até o número $2^n - 1$. Por exemplo, se $n = 4$ pode-se guardar até o 15. Não acredite em mim, faça as representações para se convencer. Se $n = 8$, guarda-se até o 255. Se $n = 16$ o limite agora é 65535, e assim por diante. Note-se que esta representação é para inteiros sem sinal (ou positivos). Se, se necessitar guardar números negativos também, perde-se um bit (o mais à esquerda) para o sinal e portanto o maior número passa a ser $2^{n-1} - 1$ e o menor, que lá no caso sem sinal era o zero, agora é -2^{n-1}. Por exemplo, se $n = 8$ o intervalo de representação é $-127..0..127$. Convenciona-se usar o bit 0 para positivo e o bit 1 para negativo e daqui:

$$\begin{array}{lcl} 11111111 & = & -(2^6 + ... + 2^1 + 2^0) = -127 \\ \ldots & & \\ 10000001 & = & \text{-1} \\ 10000000 & = & \text{-0 (?)} \\ 00000000 & = & 0 \\ \ldots & & \\ 01111111 & = & 2^6 + 2^5 + ... + 2^0 = 127 \end{array}$$

As memórias têm crescido muito nos últimos anos, e os tamanhos agora são bem generosos. Por exemplo, em C++, inteiros podem ser de 8 ou 16 bits (já visto antes) mas também de 32 bits (limite de 4.294.967.296) ou de 64 bits (limite de 18.446.744.073.709.551.616) ou a metade disso se houver sinal. A menos de algum detalhe obscuro, todas as plataformas de computadores compartilham estes valores.

Inteiros negativos

Os negativos inteiros são representados como complemento para 2. Esta operação naturalmente levará o primeiro bit a ser 1 (positivos têm primeira posição igual a zero). Mas a representação do número é um pouco diferente do que seria esperado. A regra é simples: Troque zeros por uns e vice-versa. Some 1. Pronto. Eis um exemplo: Suponha o número $(1011)_2$ como expressão de um negativo (1^o bit=1). Trocando uns-zeros fica (0100). Somando 1 fica (0101) que corresponde a 5. Como o primeiro bit era 1, então -5. Acompanhe na tabela de 4 bits:

Binário	Hexad	Decimal	Compl. 2
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Reais ou flutuantes

A dificuldade inicial é como representar o intervalo $-\infty..\infty$ em um número limitado e fixo de bits. Agora os bits disponíveis terão que ser distribuídos em diversos componentes do número a representar, a saber

$$\pm d_1 d_2 ... d_n \times \beta^{\pm e_1 e_2 ... e_m}$$

Considerando que a base β é fixa na arquitetura, ela não precisa ser guardada a cada número. Restam os dígitos $d_1 d_2 d_n$ junto com o seu sinal, além dos dígitos $e_1 e_2 ... e_m$ também com o seu sinal. O conjunto $d_1 d_2 ... d_n$ é conhecido como mantissa do número e está normalizado entre 0 e 1. Em outras palavras, o número 127.345 seria representado por 0.127345×10^3.

Padrão 16 bits original Esta representação clássica nem é mais usada, mas vai-se começar com ela por razões didáticas. Eis como funcionava um sistema em ponto flutuante, com base binária ($\beta = 2$), com $t = 10$ digitos binários na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, lembrando que $(15)_{10} = (1111)_2$ e portanto o registro é

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2	e_3	e_4

s_1 sinal do número (da mantissa). $0 \rightarrow$ positivo e $1 \rightarrow$ negativo
$d_1..d_{10}$ mantissa com 10 dígitos binários significativos
s_2 sinal do expoente: $0 \rightarrow +$ e $1 \rightarrow -$
$e_1..e_4$ expoente
Nessa representação os números são armazenados no formato normalizado (entre 0 e 1) logo, $d_1 \neq 0$ e portanto o número é

$$v = (-1)^{\pm s_1} (0.f)_2 \ 2^{\pm exp}$$

Nesse esquema, o menor valor representável é `0.1000000000.11111` e este valor é $+(0.1)_2 \times 2^{-15} = (2^{-1} \times 2^{-15})_{10} = (2^{-16})_{10} = 0.0000152587890625$.

Já o maior valor é `0.1111111111.01111` Que vem a ser $(32736)_{10} \approx 2^{15}$.

Finalmente o zero é `0.0000000000.11111`. Lembrando que este é o único número não normalizado, pois sua mantissa é toda zero. Todo número menor que o menor valor representável acarreta *underflow* enquanto todo número maior que o maior representável acarreta *overflow*.

Otimizações - padrão IEEE754

Algumas características do exposto foram alteradas para ampliar a faixa de representação e organizar a representação. Primeiro, a mantissa e o expoente têm seus locais invertidos.

Polarização A idéia é inventar um valor fixo e somá-lo a todos os expoentes, transformando um número de 4 bits mais sinal em um número de 5 bits. Neste caso (16 bits) o valor de polarização é 15 e agora passa-se de $-15..15$ a $0..31$. Isto é possível porque usando o sinal, tinham-se duas representações para o zero ($+0$ e -0) e isto é desnecessário. Agora o número é visto como

$$v = (-1)^s (0.f)_2 \ 2^{e-15}$$

Desprezo de d_1 Não se armazena o d_1 que como se viu vale sempre 1. Agora a mantissa representada passa a ter os dígitos $d_2, d_3, ...d_{t+1}$ e a posição normalizada do ponto passa a ser à direita de $d_1 = 1$ ou seja

$$v = (-1)^s (1.f)_2 \ 2^{e-15}$$

flexibilização da normalização na mantissa O explicado acima tenderia a levantar o valor da menor representação possível. Para evitar isso, quando $e = 00000$ (após a subtração de 15) a normalização do $d_1 = 1$ é eliminada. Agora o menor valor possível é `0.00000.0000000001` que é igual a `0.0009765625`. O padrão ficou assim:

s	e (5d)	f (10d)
1	2..6	7..16

O número aqui armazenado obedece a algumas regras:
se $0 < e < 31$ então $v = (-1)^s \ 2^{e-15} \ (1.f)_2$
se $e = 0$ e $f \neq 0$ então $v = (-1)^s \ 2^{-14} (0.f)$
se $e = 0$ e $f = 0$ então este número é zero
se $e = 31 = 2^5 - 1$ então v pertence à região de overflow.

Outros formatos

Esta especificação foi levada a outros formatos, a saber

A	B	C	D
formato	exp	exp_{10}	mant
binary16	5 dig	31	10 d
binary32	8 dig	255	23 d
binary64	11 dig	2047	52 d
binary128	15 dig	32767	112 d

Exemplo

Ache os valores negativos (por complemento a 2) das variáveis

```
1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1
1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0
```

São eles: -9371 e -5388

Agora, ache os flutuantes, com 7 casas decimais, de

```
1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0
0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0
0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 1
1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0
```

As respostas são -8.515625, 370.5, 1113.0000000 e -0.0588989257

Vamos ver etapa a etapa, como os resultados acima foram encontrados: Começando com o inteiro 1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 1 (caso 1, acima). Como o primeiro bit é 1, o número é negativo e precisamos invertê-lo. Fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 e agora, soma-se 1 e fica 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1. A conversão deste número para base decimal é 1+2+8+16+128+1024+8192=9371. Portanto a resposta e -9371.

Tente achar por sua conta o segundo exemplo.

Quanto ao terceiro caso, o flutuante 1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0, devemos estudar a distribuição de bits. Vamos lá

```
1 | 1 0 0 1 1 | 0 0 0 1 0 0 0 0 1 0
s   e e e e e   m m m m m m m m m m
```

O sinal do número é obtido fazendo -1^s e neste caso o número será negativo.

O expoente é -15+19=4, portanto ao final ter-se-á que multiplicar a mantissa por $2^4 = 16$

Finalmente a mantissa, é o número 0 0 0 1 0 0 0 0 1 0, que deve ter um 1 colocado à frente dando o binário de 11 bits 1 0 0 0 1 0 0 0 0 1 0. Convertendo este número a decimal tem-se 2+64+1024=1090, que deve ser dividido por $2^{11} = 2048$ dando $1090/2048 = 0.5322265625$ que multipilcado por 16 da $0.5322265625 \times 16 = 8.515625$ ou -8.515625.

Vamos ao quarto caso: 0 1 1 0 0 0 0 1 1 1 0 0 1 0 1 0 ou `0 | 1 1 0 0 0 | 0 1 1 1 0 0 1 0 1 0` O número é positivo, e seu expoente é -15+24=9. A mantissa é 1,0 1 1 1 0 0 1 0 1 0 = 1482. Ao final, o número é $+2^9 \times (1482/2048) = 370.5$.

Tente resolver agora o quinto e o sexto casos.

Como o sistema converte

Agora, a grande questão: quando você digita um número em um programa ou em uma tela, o computador precisa registrar este número na memória, usando um dos formatos aqui estudados. Vamos estudar o processo de conversão $decimal \rightarrow float$ e para campos de 16 bits. Vale a referência de este tamanho de campo não é mais usado, e só está aqui pela facilidade e simplicidade dos cálculos envolvidos. Vamos por etapas:

1. Dado um número real, por exemplo 183.27, a primeira coisa a fazer é converter este número em binário. Fica:
   $187.27_{(10)} \rightarrow 10111011.0100010100_{(2)}$

2. Agora, o ponto decimal deve ser levado à esquerda, até o primeiro 1 do número. A cada casa, o expoente deve ser incrementado. Acompanhe:
   $10111011.0100010100 \times 2^0$
   $1011101.10100010100 \times 2^1$
   $101110.110100010100 \times 2^2$
   $10111.0110100010100 \times 2^3$
   $1011.10110100010100 \times 2^4$
   $101.110110100010100 \times 2^5$
   $10.1110110100010100 \times 2^6$
   $1.01110110100010100 \times 2^7$
   $.101110110100010100 \times 2^8$
   Note que as 7 linhas acima representam o mesmo número.

3. Como o número é positivo, seu sinal é 0.

4. A potência do número é 8 que deve ser somado com 15, totalizando 23. Agora, é hora de converter o expoente em binário: $23_{(10)} \rightarrow 10111_{(2)}$

5. Finalmente, e hora de pegar 10 bits da mantissa, lembrando que o primeiro 1 (aquele que fica depois do ponto) é omitido. A mantissa fica: 0111011010

6. o numero binario em formato binary16 é
   `0 | 1 0 1 1 1 | 0 1 1 1 0 1 1 0 1 0`

7. Se você converter este número para decimal, vai encontrar 187.25, o que demonstra a baixa precisão deste formato (e explica porque ele não é usado).

☞ Para você fazer

Interprete os 2 números binários a seguir como *binary*16 inteiros e negativos e ache os valores decimais
1 e 2.

```
1 1 1 1 1 1 0 1 0 0 0 1 1 0 1 1
1 1 1 0 0 0 0 0 1 1 0 1 1 1 0 1
```

e agora interprete os 4 números a seguir como *binary*16 flutuantes e ache os decimais equivalentes com 5 casas decimais
3, 4, 5 e 6.

```
1 0 1 1 0 0 0 1 1 0 1 0 0 1 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0
0 1 1 0 1 1 1 1 0 0 0 0 1 0 0 0
0 0 1 0 1 1 1 0 1 0 1 0 1 1 0 0
```

Finalmente, converta os valores abaixo que estão em decimal para o formato *binary*16
7 e 8.

```
140
-56.98
```

Responda aqui

1.	2.		
		///// ///// /////	///// ///// /////
3.	4.	5.	6.
7			
8