README - EP1

Renan Fichberg - **NUSP:** 7991131

Laboratório de Métodos Numéricos - MAC0210 - 2016/1

Professor: Ernesto G. Birgin

Monitor: Lucas Magno

1 Arquivos

Neste primeiro exercício programa, estão sendo entregues os seguintes arquivos e diretórios:

- `/docs` - Diretório que contém este relatório.
- `/docs/relatorio` - Este documento.
- `/src` - Diretório com os códigos fonte, em GNU Octave (.m), das partes 1 e 2 especificadas no enunciado do Exercício Programa 1.
- `/docs/ieee_single.m` - Código fonte da parte 1 especificada no enunciado do Exercício Programa 1.

2 Parte 1: Representação IEEE Single

Esta seção é dedicada para falar da implementação da parte 1 descrita no enunciado do Exercício Programa 1, ressaltando os aspectos mais relevantes ou interessantes da implementação.

2.1 *Prompt: entradas*

O programa possui seu próprio terminal, por onde recebe valores numéricos do usuário e um sinal de operação. Ambos serão abordados nas próximas subseções.

2.1.1 Números

Os valores numéricos que o programa podem receber devem respeitar os formatos, em expressões regulares:

- `[0-9]+` - Números inteiros na base 10.
- `[0-9]+\.[0-9]+` - Números com ponto flutuante na base 10.
- `[0-1]+b` - Números inteiros na base 2.

- `[0-1]+\.[0-1]+b` - Números com ponto flutuante na base 2.

Assim, exemplos de entradas válidas para cada um dos items podem ser, respectivamente, 11, 5.5, 1011b e 101.1b.

Para permitir uma maior diversidade de números, o programa em nenhum instante converte as entradas para o número. Ao invés disso, ele recebe as *strings* do usuário e opera com elas mesmas. As limitações dos números são, portanto, as impostas pela própria representação IEEE Single e não pelos tipos, uma vez que as operações são realizadas em cima de cada byte (caracter) da *string* passada pelo *prompt*. A título de curiosidade, para inteiros, por exemplo, o programa não aceitará um número superior a 340282366920938463463374607431768211455 (pois qualquer número acima disso já exige um expoente E superior a 127, portanto, não encaixando nos 8 *Bits* reservados para o expoente na forma IEEE Single, que tentará guardar 127 + E no espaço de memória destinado).

Ao receber um novo número válido, o programa irá convertê-lo para o formato e imprimir seus *Bits* na tela no seguinte formato:

$$X = [b_1|b_2b_3b_4b_5b_6b_7b_8b_9|b_{10}b_{11}b_{12}b_{13}b_{14}b_{15}b_{16}b_{17}b_{18}b_{19}b_{20}b_{21}b_{22}b_{23}b_{24}b_{25}b_{26}b_{27}b_{28}b_{29}b_{30}b_{31}b_{32}]$$

Onde:

- X - A variável que representa o número passado pela linha de comando. Isso não é controlável pelo usuário e aparece junto da saída por razões meramente didáticas e ilustrativas.

- b_1 - 1 *Bit* de sinal. 0 para números positivos e 1 para números negativos.

- $b_2 ... b_9$ - 8 *Bits* do expoente. Conforme já mencionado, é escrita neste espaço de memória a *bit string* que representa o número 127 + E, onde E é o expoente do número na forma binária já normalizado (i.e, com seu *bit* oculto (= *hidden bit*) valendo 1. O *bit* oculto nada mais é que o *bit* que viria antes de b_10, imediatamente na frente do ponto flutuante).

- $b_{10} ... b_{32}$ - 23 *Bits* do significando. O significando nada mais é que os primeiros números imediatamente após o ponto flutuante.

2.1.2 Operações

As operações que podem ser realizadas no programa são apenas a soma e a subtração. Para isso, basta passar ou o caracter + ou o caracter - quando solicitado (após passar as duas entradas numéricas). As operações foram implementadas considerando 2 *bits* de guarda e 1 *sticky bit*.

Os *bits* de guarda nada mais são que 2 bits adicionais que guardam os próximos valores do significando, depois do *bit* b_{32}. Já o *sticky bit* é um bit que vem logo depois dos *bits* de guarda, que serve para avisar que ao menos um *bit* diferente de zero foi descartado ao realizar a operação de *shift* para a direita, na hora de alinhas os expoentes e os arranjar os significandos para poder realizar a operação solicitada. O *sticky bit* assumirá o valor 1 caso houve tal descarte e 0 caso contrário.

As operações são feitas do modo usual, bit-a-bit, da direita para a esquerda e utilizando *carries*.

2.2 ***Prompt:* saídas**

As saídas relativas às entradas numéricas inevitavelmente já foram cobertas na seção 2.1.1. Com relação ás operações, após passar um comando de operação válido, a saída que o usuário final recebe é o resultado da conta, com a variável "RR". RR nada mais é que uma abreviação para *Raw Result*. Tal nome foi dado pois este é o resultado sem que uma operação de arredondamento tome lugar.

2.2.1 Arredondamento

Logo após a saída RR, o programa também solicitará outras 4 saídas:

1. RD - *Round Down* - Arredondamento para $-\infty$.
2. RU - *Round Up* - Arredondamento para $+\infty$.
3. RN - *Round to Nearest* - Arredondamento para o mais próximo.
4. RZ - *Round to Zero* - Arredondamento para zero.

O modo que tais arredondamentos acontecem são descritos a seguir:

- RD - Apenas trunca o valor RR. Para todos os efeitos, RD é considerado como o menor valor mais próximo (ou igual) o valor esperado.
- RU - Soma 1 em RR, no *bit* b_{32}. Para todos os efeitos, RU é considerado como o maior valor mais próximo (ou igual) o valor esperado.
- RN - Uma vez com RU e RD, escolhe aquele que que tem o menor valor do módulo da diferente com RR.
- RZ - Considerando o sinal de RR, vai optar entre RD (se o sinal de RR for positivo) ou RU (se o sinal de RU for negativo).

Nota: os arredondamentos foram implementados desta maneira pois não foi encontrada nenhuma bibliografia que mostrasse qual é o método correto de selecionar o *rounding mode*. Assim, julguei pertinente imprimir de uma vez os quatro modos para todos os resultados.

3 Resultados da parte 1

A seguir serão mostrados os resultados e uma explicação de como o resultado foi alcançado em cada um dos exemplos do enunciado:

1) $2 + 3$

Aqui são passados os números 2, 3 e +, nesta ordem, para o programa, que imprime de saída:

- $RR = [0|10000001|01000000000000000000000]$
- $RD = [0|10000001|01000000000000000000000]$
- $RU = [0|10000001|10100000000000000000000]$
- $RN = [0|10000001|01000000000000000000000]$
- $RZ = [0|10000001|01000000000000000000000]$

Para chegar nestes valores os seguintes passos foram realizados:

1. O programa converteu o número 2 para binário.
   - $2 = (10.0)_2$
2. O programa converteu o número $(10.0)_2$ para IEEE Single.
   - Checa a posição do primeiro *bit* 1 em relação ao ponto flutuante para torná-lo o *bit* oculto.
   - É necessário fazer operações de *shift*. O expoente E assume o valor 1, pois é necessário deslocar o ponto flutuante uma posição para à esquerda para deixar o primeiro bit 1 na posição de bit oculto. Este número E então é somado a 127, e a soma (127 + 1, no caso) é convertida em uma *bit string* e armazenada nos bits $b_2 ... b_9$. É necessário que a *bit string* tenha tamanho 8, então são concatenados digitos "0"à sua esquerda até que esta tenha o tamanho esperado para ser guardada.
   - A *string* do significando precisa ter tamanho 23, então são concatenados digitos "0"à direita da *bit string* até que esta tenha o tamanho necessário.

- O número é positivo. Guardamos o sinal com o bit 0 na posição 1 do formato IEEE Single.

3. Temos que 2 = [0|10000000|00000000000000000000000]

4. Fazemos os mesmos passos com o número 3 e obtemos 3 = [0|10000000|10000000000000000000000]
   Note que o $3 = (11.0)_2$ possui dois *bits* 1, e um dos digitos portanto aparece no significando enquanto o outro torna-se o *bit* oculto, diferente do 2 que tinha apenas um *bit* 1.

5. Comparamos os expoentes e notamos que o 2 e o 3 já estão com seus expoentes alinhados. Podemos fazer a soma bit-a-bit usual (da direita para a esquerda). Ao terminá-la, obtemos 5 = [0|10000001|01000000000000000000000]. Note que os *bits* de guarda valem zero, bem como o *sticky bit*, uma vez que não houveram *shift* para a direita e portanto nenhum *bit* 1 foi descartado.

6. Agora fazemos os arredondamentos seguindo o que foi já foi explicado na seção anterior e obtemos os valores impressos acima.

2) $1 + 2^{-24}$

O programa não recebe números neste formato, portanto, precisamos reescrevê-lo para a entrada: $2^{-24} = 5.9605 \text{ x } 10^{-8} = 0.0000000059605$

Agora são passados os números 1, 0.0000000059605 e +, nesta ordem, para o programa, que imprime de saída:

- RR = [0|01111111|00000000000000000000000]
- RD = [0|01111111|00000000000000000000000]
- RU = [0|01111111|10000000000000000000000]
- RN = [0|01111111|00000000000000000000000]
- RZ = [0|01111111|00000000000000000000000]

Para chegar nestes valores os seguintes passos foram realizados:

1. O programa converteu o número 1 para binário.
   - $1 = (1.0)_2$

2. O programa converteu o número $(1.0)_2$ para IEEE Single.

- Checa a posição do primeiro *bit* 1. Como o primeiro 1 já aparece na 1ª posição da *bit string* imediatamente à esquerda do ponto flutuante, este será o *bit* oculto.
- Como não houve necessidade de fazer operações de *shift*, o expoente E assume o valor 0. Este número então é somado a 127, e a soma (127 + 0, no caso) é convertida em uma *bit string* e armazenada nos bits $b_2 ... b_9$. É necessário que a *bit string* tenha tamanho 8, então são concatenados digitos "0"à sua esquerda até que esta tenha o tamanho esperado para ser guardada.
- A *string* do significando precisa ter tamanho 23, então são concatenados digitos "0"à direita da *bit string* até que esta tenha o tamanho necessário.
- O número é positivo. Guardamos o sinal com o bit 0 na posição 1 do formato IEEE Single.

3. Temos que 1 = [0|01111111|00000000000000000000000]

4. Repetimos os passos para 0.0000000059605 = $(0.00...000110011001100110100011100...)_2$, com 28 zeros entre o primeiro *bit* 1 e o ponto flutuante. O que significa que após sucessivas vinte e oito operações de *shift*, obtemos $(1.10011001100110100011100...)_2$ x 2^{-28}. Com isso, temos nosso significando e o expoente E = -28, que será registrado na *bit string* com o valor de 127 - 28 = 99 = $(01100011)_2$. Temos, portanto, 0.0000000059605 = [0|01100011|10011001100110100011100]

5. Comparamos os expoentes e notamos que são diferentes. Alinhamos o menor (99) com o maior (127), realizando 127 - 99 = 28 operações de *shift* para a direita, a qual inevitavelmente perdemos *bits* 1, ligando portanto o *sticky bit*. Após todos os *shifts*, temos os *bits* de guarda "00". Deste modo, logo após o último *bit* do nosso significando, temos os *bits* "001".

6. Agora fazemos a soma bit-a-bit usual, da direita para a esquerda, começando na posição do *sticky bit*. Nota: concatenamos "000"à direita *bit string* do outro número para realizar a operação bit-a-bit.

7. Agora fazemos os arredondamentos seguindo o que foi já foi explicado na seção anterior e obtemos os valores impressos acima.

3) $(1.0)_2$ x 2^0 - $(1.11111111111111111111111)_2$ x 2^{-1}

O programa não recebe números neste formato, portanto, precisamos reescrever as entradas: $(1.11111111111111111111111)_2$ x 2^{-1} = $(0.111111111111111111111111)_2$ e $(1.0)_2$ x 2^0 = $(1.0)_2$

Agora são passados os números 1b, 0.111111111111111111111111b e a operação -, nesta ordem, para o programa, que imprime de saída:

- RR = [0|01100111|00000000000000000000000]
- RD = [0|01100111|00000000000000000000000]
- RU = [0|01100111|10000000000000000000000]
- RN = [0|01100111|00000000000000000000000]
- RZ = [0|01100111|00000000000000000000000]

Para chegar nestes valores os seguintes passos foram realizados:

1. O programa aproveita as *strings* binárias que lhe foram passadas. Não há necessidade de realizar conversões.
2. O programa converteu o número $(1.0)_2$ para IEEE Single. O resultado é o mesmo que 1 na base 10 do exemplo anterior. Os mesmos passos foram realizados.
3. Repetimos os passos para $(0.111111111111111111111111)_2$. É necessário fazer uma operação de *shift* para esquerda, nos deixando com o expoente -1, que será registrado como 127 - 1 = 126 = $(01111110)_2$. O significando, por sua vez, cabe exatamente, e portanto os *bits* de guarda e o *sticky bit* permanecem como "000"imediatamente após a *bit string* do significando. Temos assim $0.111111111111111111111111b = [0|01111110|11111111111111111111111]$
4. Comparamos os expoentes e notamos que são diferentes. Alinhamos o menor (126) com o maior (127), realizando 127 - 126 = 1 operação de *shift* para a direita. Conforme mencionado no item anterior, não há perda de *bits* e o significando cabe exatamente no espaço de memória a ele destinado.
5. Agora fazemos a subtração bit-a-bit usual, da direita para a esquerda.
6. Em seguida fazemos os arredondamentos seguindo o que foi já foi explicado na seção anterior e obtemos os valores impressos acima.

4) 1.0 - $(1.00000000000000000000001)_2$ x 2^{-25}

O programa não recebe números neste formato, portanto, precisamos reescrever a entrada: $(1.00000000000000000000001)_2$ x $2^{-25} = (0.000000000000000000000000100000000000000000000001)_2$

Agora são passados os números 1, 0.000000000000000000000000100000000000000000000001b e a operação -, nesta ordem, para o programa, que imprime de saída:

- RR = [0|01111110|11111111111111111111111]

- RD = [0|01111110|11111111111111111111111]

- RU = [0|01111111|00000000000000000000000]

- RN = [0|01111110|11111111111111111111111]

- RZ = [0|01111110|11111111111111111111111]

Para chegar nestes valores os seguintes passos foram realizados:

1. O programa aproveita as *strings* binárias que lhe foram passadas. Não há necessidade de realizar conversões.

2. O número 1 é feito do mesmo modo que foi explicado nos exemplos anteriores.

3. Com relação ao número $(0.0000000000000000000000001000000000000000000000001)_2$, precisamos realizar 25 operações de *shift* para a esquerda até que o primeiro *bit* 1 fique na posição do *hidden bit*, o que nos resultará ao final um expoente de -25, que será registrado como 127 + (-25) = 102 = $(01100110)_2$. Já o significando será a *bit string* "00000000000000000000001", com *bits* de guarda e o *sticky bit* = "000"(pois foram *shifts* para a esquerda e o 1 foi o último caracter passado, não sendo portanto um número que continua). Terminamos, finalmente, com $STRING$ = [0|01100110|00000000000000000000001], onde "STRING"é a segunda entrada numérica.

4. Comparamos os expoentes e notamos que são diferentes. Alinhamos o menor (102) com o maior (127), realizando 127 - 102 = 25 operação de *shift* para a direita. Conforme mencionado no item anterior, não há perda de *bits* e o significando cabe exatamente no espaço de memória a ele destinado.

5. Agora fazemos a subtração bit-a-bit usual, da direita para a esquerda.

6. Em seguida fazemos os arredondamentos seguindo o que foi já foi explicado na seção anterior e obtemos os valores impressos acima.

4 Observações Finais da parte 1

São considerados **apenas** números *normalizados* na entrada e nos resultados, portanto, números como o zero estão fora do escopo da implementação. Tentar forçar operações a resultar em números que seriam representados como subnormais pode (e deve) resultar em um comportamento não esperado.

Os números só serão reconhecidos nos **formatos apontados** nesta documentação. Entradas como 1x2ˆ(-10) não funcionarão.