Arquitectura de Computadores

Representação em Vírgula Flutuante

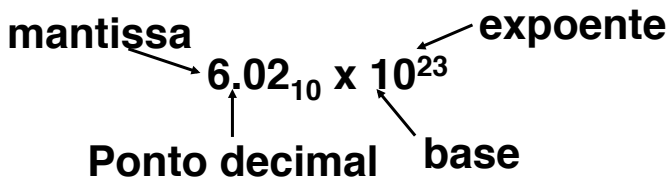
Docente: Pedro Sobral
http://www.ufp.pt/~pmsobral

Representando valores não inteiros

- Como representar?
  - Números muito grandes?(segundos/século)
    $3{,}155{,}760{,}000_{10}$ ($3.15576_{10} \times 10^{9}$)
  - Muito pequenos? (diametro de um átomo)
    0.00000001_{10} ($1.0_{10} \times 10^{-8}$)
  - Racionais (dízimas infinitas)
    2/3 (0.666666666. . .)
  - Irracionais
    $2^{1/2}$ (1.414213562373. . .)
  - Transcendentes
    e (2.718...), π (3.141...)

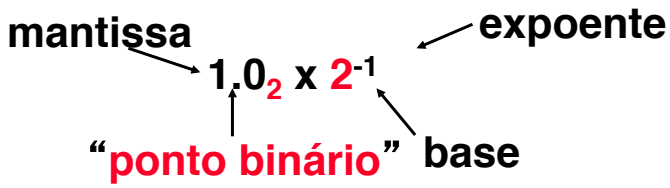
Todos representados em notação científica!

Notação Científica (em decimal)



- Forma Normalizada: não representar 0s à frente (exactamente 1 dígito à esquerda do ponto decimal)
- Alternativas para representar 1/1,000,000,000
  - Normalizada: 1.0×10^{-9}
  - Não normalizada: 0.1×10^{-8},10.0×10^{-10}

Notação Científica (em binário)



- A aritmética no computador que suporta esta representação tem o nome de vírgula flutuante, uma vez que representa números onde o ponto binário não é fixo, como é o caso dos inteiros.
  - Variáveis declaradas em C como `float`

Representação em Vírgula Flutuante (1/2)

- Formato Normal: $+1.xxxxxxxxxx_2 * 2^{yyyy_2}$
- Representando numa palavra (32 bits)

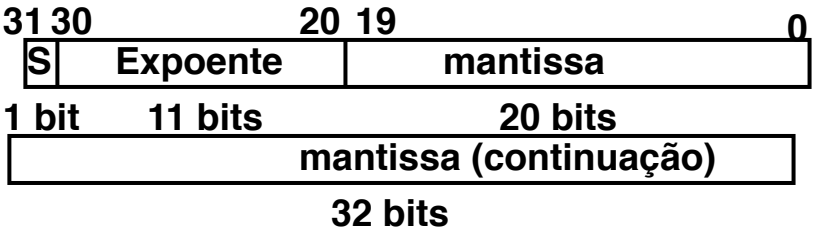
- S representa o Sinal, Expoente representado pelos y's, Mantissa pelos x's
- Representa números desde 2.0×10^{-38} até 2.0×10^{38}

Representação em Vírgula Flutuante (2/2)

- E se o resultado for muito grande?($> 2.0 \times 10^{38}$)
  - "Overflow"!
  - "Overflow" ⇒ Expoente impossível de representar com 8-bit
- E se for muito pequeno? (>0, $< 2.0 \times 10^{-38}$)
  - "Underflow"!
  - "Underflow" ⇒ Expoente negativo impossível de representar com 8-bit
- Como reduzir a probabilidade de "Overflow" e "Underflow"?

Rep. Vírg. Flutuante em Precisão Dupla

° Número representado em 2 palavras (64 bit)



° Precisão dupla (vs. Precisão simples)

- Em C declarar a variável como `double`
- Representa números desde 2.0 x 10^{-308} até 2.0 x 10^{308}
- Mas a principal vantagem é uma maior precisão devido ao número de bits da mantissa.

A Norma IEEE 754 (1/5)

° Precisão simples e dupla semelhantes

° Bit de sinal: 1 negativo
0 positivo

° Mantissa:

- Para aumentar a precisão, o 1 antes do ponto binário é implicito e não se representa
- 1 + 23 bits simples, 1 + 52 bits dupla
- Sempre verdade: Mantissa < 1 (para números Normalizados)

° Nota: 0 não tem 1 antes do ponto binário, portanto reserva-se o expoente 0 para o número 0.

A Norma IEEE 754 (2/5)

- A ideia era usar números em VF mesmo que não existisse hardware dedicado
- Podemos comparar números em VF em 3 etapas: Comparar os sinais, depois os expoentes e finalmente as mantissas.
- A comparação deveria ser rápida, especialmente para números positivos
- Portanto:
  - Comparar sinais (negativo < positivo)
  - Comparar expoentes, (maior exp. => maior #)
  - Por fim a mantissa: (exp. Iguais, maior mantissa => maior #

A Norma IEEE 754 (3/5)

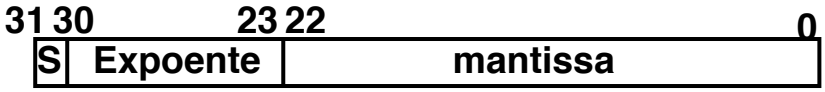
- Expoente Negativo?
  - Complem. Para 2? 1.0 x 2^{-1} v. 1.0 x2^{+1} (1/2 v. 2)

1/2	0	1111 1111	000 0000 0000 0000 0000 0000
2	0	0000 0001	000 0000 0000 0000 0000 0000

  - Nesta notação usando comparação de inteiros temos 1/2 > 2!
- Alteração: usar uma nova notação para representar o expoente…

A Norma IEEE 754 (4/5)

- Tem o nome de notação desviada "Biased Notation", onde o desvio é o número a subtrair ao expoente para obter o valor real
  - IEEE 754 usa desvio 127 para precisão simples e 1023 para precisão dupla, ou seja:
- Para precisão simples….
  - Subtrair 127 ao campo do expoente para obter o valor real do expoente.



1 bit 8 bits 23 bits

- $(-1)^S \times (1 + \text{mantissa}) \times 2^{(\text{Expoente}-127)}$
  - Para precisão dupla é idêntico excepto que o desvio é 1023

A Norma IEEE 754 (5/5)

- Como fica então 1.0×2^{-1} v. $1.0 \times 2^{+1}$ (1/2 v. 2) ?
  - Expoente=valor+127
  - (1/2) = 2^{-1} temos $-1+127=126_{10}=01111110_2$
  - (2)= 2^{+1} temos $+1+127=128_{10}=10000000_2$

1/2	0	0111 1110	000 0000 0000 0000 0000 0000
2	0	1000 0000	000 0000 0000 0000 0000 0000

"Pai" da representação em V. Flutuante

Norma IEEE 754 para a representação binária em vírgula flutuante

Prof. Kahan

1989 ACM Turing Award Winner!

`www.cs.berkeley.edu/~wkahan/`
`…/ieee754status/754story.html`

Converter: Vírgula Flutuante para Decimal

0	0110 1000	101 0101 0100 0011 0100 0010

- Sinal: 0 => positivo
- Expoente:
  - $0110\ 1000_2 = 104_{10}$
  - Ajustando o desvio: 104 - 127 = -23
- Mantissa:
  - $1 + 1x2^{-1} + 0x2^{-2} + 1x2^{-3} + 0x2^{-4} + 1x2^{-5} + \ldots$
    $= 1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-22}$
    $= 1.0_{10} + 0.666115_{10}$

Representa: $1.66611510 * 2^{-23} \sim 1.986 * 10^{-7}$

Converter: Decimal para Vírgula Flutuante (1/2)

- 23,625
- Parte inteira: $23_{10}=10111_2$
- Parte fraccionária: 0.625
  - 0.625*2=1.25 — 0.625=0.1…
  - 0.25*2=0.5 — 0,625=0.10…
  - 0.5*2=1.0 — 0,625=0.101…
  - 0*2=0.0 — 0,625=0.1010…
- Quando parar?
  - Quando o resultado do cálculo for zero, pois todos os cálculos seguintes são zero, ou
  - Quando o número de bits da mantissa for atingido

Converter: Decimal para Vírgula Flutuante (2/2)

- 23.625
  - 10111.101000....
  - $1.0111101000....x2^4$
- Temos então:
  - Sinal positivo: 0
  - Expoente: 4+127=131=10000011
  - Mantissa: 01111010000000000000000

0	10000011	01111010000000000000000

0x41BD0000

Exercício

- Converta para decimal o seguinte número representado em VF
  - a) 0x41440000
- Converta para VF (em Hex) o seguinte decimal
  - b) 11.4375

a) 12,25

b) 0x41370000

Representando ± ∞

- Em VF, a divisão por 0 deve produzir ± ∞, e não "overflow".
- Porquê?
  - É possível fazer operações com ∞….. Exemplo: X/0 > Y pode ser uma comparação válida
  - Lembra-se do cálculo dos limites…
- IEEE 754 representa ± ∞
  - O maior expoente (255) reservado para ∞
  - Mantissa toda a zeros

Representando 0

- Como representar o 0?
  - Expoente todo a zeros
  - Mantissa também a zeros
  - E o sinal?
  - +0: 0 00000000 00000000000000000000000
  - -0: 1 00000000 00000000000000000000000
- Porquê dois Zeros?
  - Lembra-se dos limites?
  - Importante em cálculos matemáticos

Números especiais

- O que definimos até agora? (Precisão simples)

Expoente	mantissa	Objecto
0	0	0
0	não zero	???
1-254	qualquer	+/- nº em VF
255	0	+/- ∞
255	não zero	???

- Vamos aproveitar o que falta…
  - Exp=0|255 & mantissa!=0 …

Representando NaN "Not a Number"

- O que é `sqrt(-4.0)` ou `0/0`?
  - Se ∞ não é erro, estes também não devem ser.
  - Têm o nome de "Not a Number" (NaN)
  - Expoente = 255, Mantissa !=0
- Para que servem?
  - Espera-se que ajudem a depurar erros…
  - Contaminam os cálculos: qualquer operação com NaN tem sempre NaN como resultado…

Números não normalizados (1/2)

- Problema: Há uma falha entre os números representáveis em VF à volta de 0…
  - Qual o número mais pequeno:

    $a = 1.0..._2 * 2^{-126} = 2^{-126}$
  - E o seguinte:

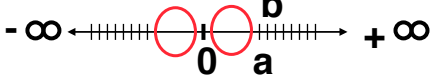
    $b = 1.000.....1_2 * 2^{-126} = 2^{-126} + 2^{-149}$

$a - 0 = 2^{-126}$

$b - a = 2^{-149}$

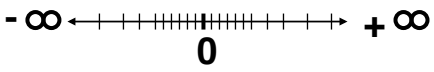
A culpa é da normalização e do 1 implícito!

Falhas!



Números não normalizados (2/2)

- Solução:
  - Ainda não usamos o Expoente = 0, e a mantissa != 0
  - Número não normalizado:
    - não tem 1 implícito,
    - tem um expoente implícito = -126.
  - Número mais pequeno: a = 2^{-149}
  - E o seguinte: b = 2^{-148}



Arredondamento

- É necessário proceder ao arredondamento da mantissa de forma que o resultado se ajuste ao número de bits disponíveis.
- O hardware de VF possui 2 bits de precisão extra e arredonda os resultados convenientemente.
- O arredondamento ocorre na conversão…
  - De precisão dupla para simples
  - De VF para inteiro

IEEE: 4 modos de arredondamento

- Arredondar para + ∞
  - SEMPRE para "cima": 2.1 ⇒ 3, -2.1 ⇒ -2
- Arredondar para - ∞
  - SEMPRE para "baixo": 1.9 ⇒ 1, -1.9 ⇒ -2
- Truncar
  - "Esquecer"os últimos bits (arredondar para 0)
- Arredondar para o par mais próximo
  - 2.5 ⇒ 2, 3.5 ⇒ 4
  - Abordagem equilibrada
  - Metade das vezes arredondamos para cima e outra metade para baixo

Multiplicação de Inteiros (1/2)

- No MIPS multiplicamos registos, logo:
  - Nº 32-bit x Nº de 32-bit = Nº 64-bit
- Sintaxe da multiplicação (com sinal):
  - `mult` registo1, registo2
  - Multiplica os valores de 32-bit nestes registos e coloca o resultado de 64-bit em 2 registos especiais:
    - Os 32-bit mais significativos no hi, e os 32-bit menos significativos no lo
  - O hi e o lo são dois registos não incluidos nos 32 que conhecemos…
  - Use `mfhi` registo & `mflo` registo para mover do hi e lo para outro registo.

Multiplicação de Inteiros (2/2)

- **Exemplo:**
  - em C: `a = b * c;`
  - No MIPS:
    - *b* está em $s2; *c* em $s3; e *a* em $s0 e $s1 (pode ter 64 bits)

```
mult $s2,$s3   # b*c
mfhi $s0       # Metade superior
               # do produto em $s0
mflo $s1       # E a inferior
               # em $s1
```

- **Nota: Geralmente só nos interessa a metade inferior.**

Divisão de Inteiros

- **Sintaxe da divisão (com sinal):**
  - `div` registo1, registo2
  - Divide o registo1 pelo registo2 (32-bit):
  - O resto fica no `hi`, o quociente no `lo`
- **Implementa em C a divisão (`/`) e módulo (`%`)**
- **Exemplo em C:**

```
a = c / d;
b = c % d;
```

- **No MIPS:** `a↔$s0;b↔$s1;c↔$s2;d↔$s3`

```
div  $s2,$s3    # lo=c/d, hi=c%d
mflo $s0        # quociente
mfhi $s1        # resto
```

Operandos sem sinal & "Overflow"

- O MIPS tem versões do `mult`, `div` para operandos sem sinal:

  `multu`

  `divu`

- MIPS não verifica a ocorrência de "overflow" em NENHUMA instrução de multiplicação ou divisão
  - Cabe ao software verificar o `hi`

Somar e Subtrair em VF

- Muito mais difícil do que com inteiros (não basta somar as mantissas…)
- Adição:como se faz?
  - "des"normalizar para o maior expoente
  - Adicionar as mantissas
  - Normalizar (& verificar "under/overflow")
  - Arredondar se necessário (pode ser necessário renormalizar)
- Subtracção: é semelhante…
- Pergunta: Como se integra isto na unidade de lógica e aritmética de inteiros? [Resposta: Não se integra!]

Arq. Vírgula Flutuante do MIPS (1/4)

- Instruções separadas para VF:
  - Precisão simples:
    `add.s, sub.s, mul.s, div.s`
  - Precisão dupla:
    `add.d, sub.d, mul.d, div.d`
- Estas são DE LONGE mais complexas do as correspondentes para inteiros
  - Levam muito mais tempo a executar

Arq. Vírgula Flutuante do MIPS (2/4)

- Problemas:
  - É pouco eficiente ter instruções que têm tempos de execução radicalmente diferentes no processador.
  - Geralmente, um valor não muda de VF ⇔ `int` durante um programa.
    - Só um tipo de instrução é usado neste valor
  - Alguns programas não usam VF
  - É necessário muito hardware relativo a inteiros para trabalhar em VF rapidamente

Arq. Vírgula Flutuante do MIPS (3/4)

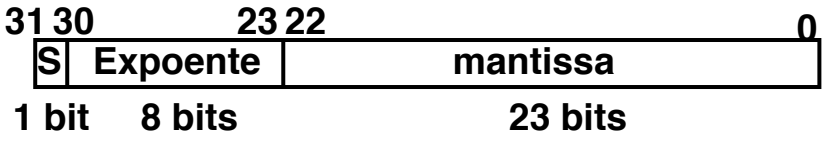
- Solução em 1990: Fazer um "Chip" completamente independente que apenas trate da VF.
- Coprocessador 1: "chip" VF
  - contém 32 registos de 32-bit: `$f0`, `$f1`, …
  - Os registos são usados nas instruções `.s` e `.d`
  - Instruções novas para aceder à memória: `lwc1` e `swc1` ("load word coprocessor 1", "store …")
  - Precisão dupla: Por convenção, 2 registos consecutivos Par/ímpar contém um Nº em VF de PD : `$f0`/`$f1`, `$f2`/`$f3`, … , `$f30`/`$f31`
    - O registo par é que dá o nome

Arq. Vírgula Flutuante do MIPS (4/4)

- Computador contém co-processadores:
  - Processador: Trata das instruções "normais"
  - Coprocessador 1: VF e só VF;
  - Mais coprocessadores?… sim, mais tarde
  - Hoje, coprocessador de VF está integrado com o CPU,
  - Há "chips" mais baratos que não têm HW de VF
- Instruções para mover dados entre o processador e o coprocessador:
  - `mfc0`, `mtc0`, `mfc1`, `mtc1`, etc.
- Mais instruções na aula prática…

Em conclusão…(1/2)

- Números em vírgula flutuante aproximam os valores que queremos usar.
- A norma IEEE754 é a abordagem mais usada para a interpretação desses números
  - Todos os PCs e servidores vendidos depois de ~1997 seguem esta norma
- Sumário (Precisão simples):



- $(-1)^S \times (1 + \text{Mantissa}) \times 2^{(\text{Expoente}-127)}$
  - Precisão dupla é semelhante com desvio 1023

Em conclusão…(2/2)

- Vírgula Flutuante:

Expoente	Mantissa	Objecto
0	0	0
0	não zero	Desnorm
1-254	qualquer	+/- Nº em VF
255	0	+/- ∞
255	não zero	NaN

- `mult`, `div` usam os reg. `hi`, `lo`
  - `mfhi` e `mflo` para copiar.
- O Arredondamento é feito para o par mais próximo (por defeito)