Universidade de Brasília

Organização e Arquitetura de Computadores

Experimento II

Aritimética Computacional Fracionária

---

19 de Maio de 2014

Professor Marcus Vinícius Lamar - turma A

Alunos:

Danilo Cardoso	11/0010515
Gustavo Meneses	10/0103766
Juarez Aires	11/0032829
Matheus Ramiro	09/0125894

A Aritimética de Ponto Flutuante - Software

A.1 Raízes da Equação de Segundo Grau

Consideramos um polinômio $p(x)$ de segunda ordem dado por:

$$p(x) = ax^2 + bx + c \tag{1}$$

Queremos achar as raízes r_1 e $r_2$de $p(x)$. Isso foi feito utilizando-se a fórmula de Bhaskara:

$$r_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{2}$$

se $b^2 - 4ac \geq 0$ e

$$r_{1,2} = \frac{-b \pm j\sqrt{-(b^2 - 4ac)}}{2a} \tag{3}$$

se $b^2 - 4ac < 0$.

Vemos que a aplicação da fórmula requer uma rotina para cálculo de raíz quadrada positiva. Quando fizemos o trabalho não sabíamos que o MIPS já possuia uma rotina para essa finalidade e implementou-se então o método da bisseção descrito a seguir:

```
recebe (X,e)
L = 0
R = X
E = R - L
enquanto E > e
    M = (L+R)/2
    se M^2 > x, então R = M
    se M^2 < x, então L = M
    se M^2 == x, então retorne M
fim enquanto
retorna M
```

Em nossa implementação definimos 'e' fixo e igual a .Além disso, para melhor precisão, todos os cálculos foram feitos com double e só ao final convertidos para float de precisão simples.

A.1.a O cógido em assembly MIPS

O código completo desenvolvido pode ser visto aqui.

A.1.b O procedimento show

O procedimento show pode ser visto no mesmo link acima.

A.1.c Resultados

O código é testado com algumas entradas e a resposta mostrada na tabela a seguir:

Entrada			Saída	
a	b	c	r_1	r_2
1	0	-9.86960440	3.1415925	-3.1415925
1	0	0	0.0	-0.0
1	99	2459	-49.5 + i* 2.95804	-49.5 + i* -2.95804
1	-2468	33762440	1234.0 + i* 5678.0	234.0 + i* -5678.0
0	10	100	-10	NaN

Tabela 1: testes da implementação

A.2 Casas Decimais de π

Para calcular os dígitos de pi, usaremos o *Spigot Algorithm for the Digits of Pi*[1] descrito em [2]. A seguir apresentamos brevemente a motivação teórica para o algoritmo e em seguida apresentamos o algoritmo propriamente dito.

Primeiramente faremos uma reinterpreação das casas decimais:

$$\sqrt{2} = 1.41421356 = 1+\frac{1}{10}\left(4+\frac{1}{10}\left(1+\frac{1}{10}\left(4+\frac{1}{10}\left(2+\frac{1}{10}\left(1+\ldots\right)\right)\right)\right)\right) \tag{4}$$

Podemos resumir a notação dizendo que $\sqrt{2} = (1;4,1,4,2,1,\ldots)_d$, onde d é a base decimal $\{\frac{1}{10},\frac{1}{10},\frac{1}{10},\frac{1}{10},\ldots\}$. A base d é aquela que estamos mais habituados, mas qualquer coleção de frações pode formar uma base nesse sentido e todo número terá uma representação nessa base criada. Na base binária, por exemplo, apenas substituímos o 10 do denominador por 2. Veja que na base decimal e na base binária, temos os mesmos elementos em cada entrada do vetor base, e se isso não acontecer?

Como um exemplo de uma base exótica e de aplicação interessante, considere a base $b = \{\frac{1}{2},\frac{1}{3},\frac{1}{4},\frac{1}{5},\ldots,\frac{1}{n}\}$. Para ver como ela pode ser útil, utilize a série de potências da exponencial para escrever o número de Euler e como:

$$exp(1) = e = \sum_{i=0}^{\infty}\frac{1^i}{i!} = \sum_{i=0}^{\infty}\frac{1}{i!} = 1+\frac{1}{1}+\frac{1}{2}+\frac{1}{3*2}+\ldots \tag{5}$$

que pode ser reescrito como:

$$e = 1+\frac{1}{1}(1+\frac{1}{2}(1+\frac{1}{3}(1+\frac{1}{4}(1+\frac{1}{5}(1+\ldots))))) \tag{6}$$

[1] em uma tradução livre: Algoritmo Torneira para os Dígitos de Pi

ou seja,:

$$e = (2; 1, 1, 1, 1, 1, 1, \ldots)_b \tag{7}$$

isto é, o número e é uma dízima periódica na base b! Será que não podemos achar uma base que represente π em uma dízima periódica? É justamente essa a ideia do algoritmo utilizado. No caso, a base utilizada é $c = \{\frac{1}{3}, \frac{2}{5}, \frac{3}{7}, \frac{4}{9}, \ldots, \frac{i-1}{2i-1}, \ldots\}$ e é possível mostrar, também utilizando séries de potência, que:

$$\pi = (2; 2, 2, 2, 2, 2, 2, 2, \ldots)_c \tag{8}$$

O trabalho para achar as casas decimais de pi consiste então em converter a representação na base c para a base decimal usual! O algoritmo é apresentado a seguir:

```
1. Inicialize: Seja A = (2,2,2,2,2,....,2) um array de comprimento
[10n/3] + 1
2. Repite n + 1 vezes
    2.1: multiplica A por 10
    2.2: coloque na forma regular:
        comecando da direita ate o segundo elemento do vetor:
            reduza o i-esimo elemento mod(2i-1), isto é:
             calcule q e r tal que: A[i] = q*(2i-1) + r
             A[i] = r
             A[i-1] += q(i-1)
    2.3: calcule o proximo pre-digito
           reduza o primeiro elemento mod(10), isto é:
             A[0] = q*(10) + r
             A[0] = r
             salve q para a próxima etapa
    2.4: decida o que fazer com o vetor preDigitos com base em q:
            se q == 9:
                preDigitos.add(q)
            caso contrario, se q == 10:
                q = 0
                adicione 1 a todos os pre-digitos anteriores( 9 vira 0)
                digitosVerdadeiros.add(preDigitos)
                esvazia preDigitos
                adiciona q aos preDigitos
             caso contrario:
                digitosVerdadeiros.add(preDigitos)
                esvazia preDigitos
                adiciona q aos preDigitos
```

Figura 1: Algoritmo π-spigot

Note que **o algoritmo usa apenas aritmética de interiso para calcular os elementos de** π e é de simples implementação computacional!

No trabalho desenvolvido, o algoritmo foi primeiramente implementado em C++ para testar sua funcionalidade e em seguida implementado em assembly para MIPS. Analisando o algoritmo, vemos que ele precisa de estruturas de dados para armazenar os números já calculados e outra para armazenar os pre-dígitos, que deve ter o tamanho acrescentado e resetado ao longo do algoritmo. Boa parte do trabalho em assembly foi implementar essa estrutura de dados para lidar com dados de tamanho de variado.Em C++ utilizamos a estrutura *vector* que já está implementada e isso facilita muito a implementação. Vale a pena comparar o tamanho do código: em C++ foram necessárias 109 linhas, em assembly o código completo usa 740. Além disso, o código em C++ foi escrito em torno de 30 minutos e o código em assembly precisou de quase 5 horas.

A.2.a O cógido em assembly MIPS

O código completo desenvolvido em C++ pode ser visto aqui e o desenvolvido em assemlby MIPS pode ser visto aqui.

A.2.b A função show_pi

A função pode ser vista no mesmo código assembly mips indicado anteriormente.

A.2.c O Custo Computacional

O programa foi rodado diversas vezes para valores diferentes do número de casas desejadas e o número de operações necessárias foi anotado. Obteve-se a seguinte tabela:

número de casas	número de instruções
5	19839
10	70565
15	152361
20	261474
25	404655
30	578862

Tabela 2: Número de instruções por número de casas calculadas para o Spigot Algorithm

A seguir vemos os dados plotados em um gráfico junto com uma regressão polinomial de segunda ordem.

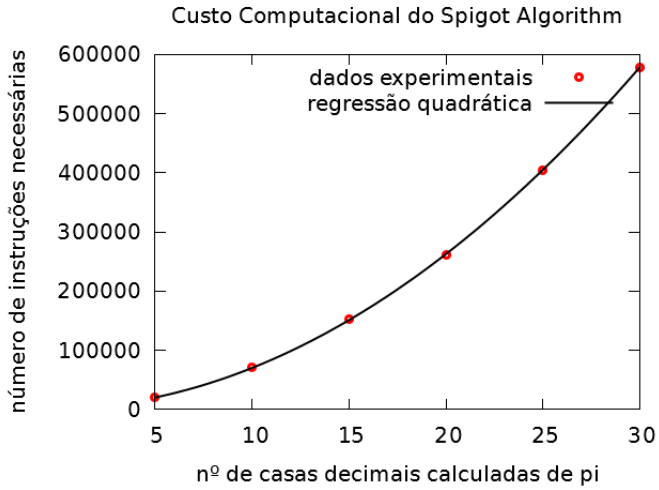

Figura 2: Dados e Regressão para o custo do Spigot Algorithm

O polinômio calculado para o custo é:

$$c(n) = 616.4 \cdot n^2 + 749.2 \cdot n + 1133.7 \tag{9}$$

A extrapolação para n = 100000 é mostrada a seguir:

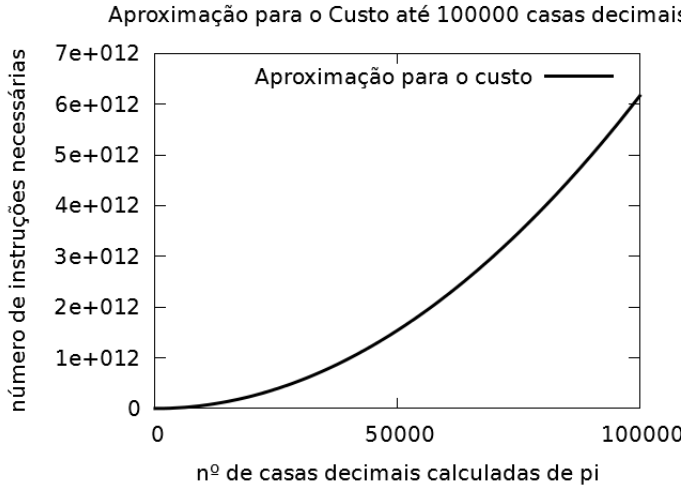

Figura 3: Extrapolação para o custo do Spigot Algorithm

A.2.d Limitações no MARS

O fator limitante para grandes valores de n é o tamanho da memória. O algoritmo utiliza 3 vetores: **A**, **preDigits** e **piDigits**. **A** tem tamanho $\frac{10}{3}n + 1$, **preDigits** e **piDigits** tem tamanho n. Além disso, cada dígito

do vetor é um número de 0 a 9, então pode ser armazenado em 1 byte. O algoritmo precisa então de:

$$m(n) = \frac{10}{3}n + 1 + 2n = \frac{19}{3} \cdot n \text{ bytes} \tag{10}$$

Estamos utilizando o recurso de armazenamento de dados na heap por meio do syscall 9 que aloca memória dinamicamente. Para tanto, o MARS disponibiliza os endereços de 0x10040000 até (0x100401e0 + 1c). Contando em words isso dá 8x16 words (8 colunas por 16 linhas). Portanto, o número de bytes disponíveis para acesso dinâmico é 8x16x4 = 512 bytes. Basta então resolver para n:

$$\frac{19}{3}n = 512 \Rightarrow n = 80.8421 \tag{11}$$

Portanto, com o MARS podemos calcular no máximo 80 dígitos de pi devido à limitação de memória

A.2.e Tempo de Execução para n máximo

Como já dissemos, **nosso algoritmo não utiliza aritimética de ponto flutuante**, por isso levamos em conta apenas o custo das operações de ponto fixo. Jogamos o n máximo na fórmula obtida por regressão para obter o número de instruções necessárias para n = 80:

$$c(96) = 616.4 \cdot 80^2 + 749.2 \cdot 80 + 1133.7 = 4.0060 \cdot 10^6 \text{ instruções} \tag{12}$$

Calculamos o tempo de ciclo:

$$T = \frac{1}{f} = \frac{1}{1E9} = 1 \cdot 10^{-9} \text{ segundos} \tag{13}$$

Utilizamos a fórmula clássica para tempo de execução:

$$t = n \cdot CPI \cdot T = 4.0060E06 \cdot 1 \cdot 1E{-}9 = 4.0060 \cdot 10^{-3} s = 4.0060 \text{ milisegundos} \tag{14}$$

A.2.f Recorde Atual de casas para Pi

Segundo a wikipedia, em 2011 já havia-se calculado π com mais de 10 trilhões de casas decimais

B Aritimética de Ponto Flutuante - Hardware

B.1 ULA de Inteiros

B.1.a Análise de Funções

A ULA implementada é uma versão bem mais completa do que a vista em sala, com um total de 18 operações. A ULA é uma parte do circuito que tem apenas dois registradores internos, HI e LO e os utiliza para operações de multiplicação e divisão (devido ao fato de essas operações causarem overflow frequentemente). Todas as outras operações têm seus resultados salvos ou no banco de registradores ou na memória. Os componentes da ULA são:

- A e B, números de 32 bits que servem como entrada.
- Sinal de controle, parâmetro que escolhe a operação a ser realizada.
- ALUresult, o resultado da operação.
- Shamt, um parâmetro importante para as operações de sll, srl, e sra, já que nesse número consta o argumento da operação.
- Zero e Overflow, duas flags que são ligadas caso o resultado da operação atenda aos requisitos.
- Relógio para habilitar a operação e manter o sistema síncrono.
- Reset para zerar a memória da ULA (registradores HI e LO).

Para manter a tabela o mais limpa possível, apenas selecionamos as operações e colocamos os resultados finais, sem se preocupar com as flags, memória da ULA e casos específicos.

Sinal de Controle	Operação	ALUresult
0	AND	A AND B
1	OR	A OR B
2	ADD	A + B
3	MFHI	HI
4	SLL	B « shamt
5	MFLO	LO
6	SUB	A-B
7	SLT	A<B: 0 else: 1
8	SRL	B » shamt
9	SRA	B » shamt (si gn extended)
10	XOR	A XOR B
11	SLTU	A < B
12	NOR	A NOR B
13	MULT	A * B
14	DIV	A / B
15	LUI	B « 16
16	SLLV	B « A
17	SRAV	B » A (sign extended)
18	SRLV	B » A
Outro	Padrão	0

Tabela 3: Opcode, operação e resultado

Casos especiais:

- mult e div colocam seu resultado tanto em hi/lo quanto na saída, mas o controle não cede permissão de escrita no banco de registradores da CPU com os resultados da operação, fazendo com que seja necessárias as instruções mfhi e mflo.
- Zero é uma flag que ativa sempre que todos os bits do resultado da operação são nulos.
- Overflow é uma flag que ativa quando o sinal do resultado não coincide com o sinal que é esperado devido à operação e os sinais dos dados de entrada. Essa é uma propriedade dos números em complemento de dois que facilita e detecção do overflow.

B.1.b Verificação

Para s simulação, foi feita uma pequena quantidade de operações na ULA para verificar os resultados e confirmar seu funcionamento adequado. Testamos todas as operações e os resultados condizem com o esperado.

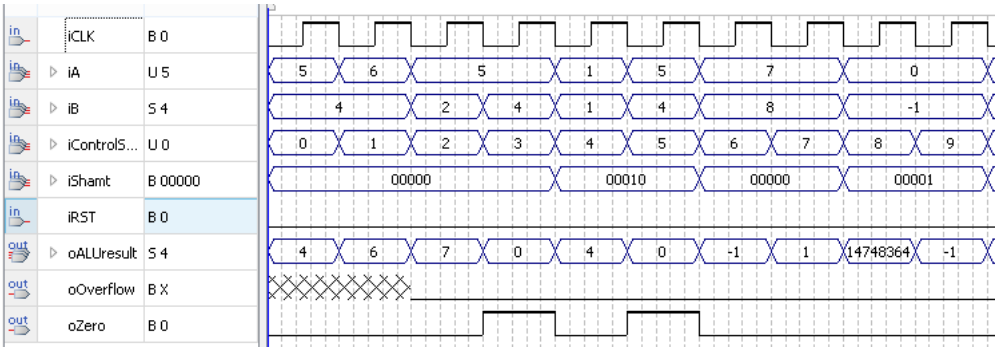

Figura 4: Forma de onda de teste: entrada e saída

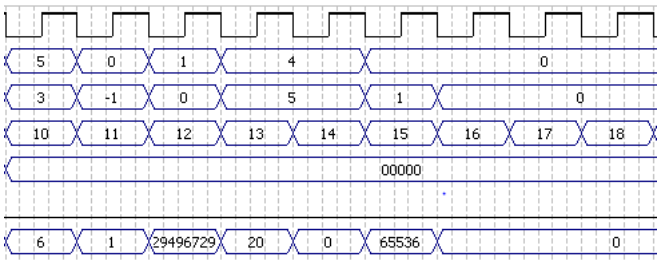

Figura 5: continuação

Todas as operações fazem sentido, já que os registradores inicializam vazios e, portanto as operações mfhi e mflo vão resultar em 0 antes de algum comando mult ou div.

B.1.c Sintetizando na FPGA

Foi feito o teste do sistema na FPGA e de fato, foi verificado que as operações da ULA funcionam bem. Devido à pequena quantidade de entradas (temos 18 switches e 4 botões enquanto precisaríamos de 69 formas de entrada apenas para ter os dois argumentos e o controle), para efeito de teste usamos entradas de 5 bits cada uma e mais o controle de 5 bits também.

O link para o vídeo do teste pode ser visto aqui.

B.1.d Requisitos

Fazendo uma análise pelo timequest obtemos os seguintes parâmetros de desempenho:

- Numero de elementos lógicos: 3945
- Frequência máxima: 6.28 MHz
- Setup Time: 153.18 ns

- Hold Time: -6.061 ns (negativo? Segundo o google, é possível.)
- Propagation Delay: 159.18 ns

B.2 Banco de Registradores da CPU Principal

B.2.a Construção

O banco de registradores é a memória mais rápida e cara de um processador. A arquitetura mips tem um banco de 32 registradores de 4 bytes cada um e esses registradores tem função de leitura e escrita. A implementação pode ser feita como um módulo, retirando as responsabilidades de controle e seleção dos registradores para um módulo de controle separado e deixando para esse módulo apenas a leitura e escrita na sua memória interna.

Figura 6: banco de registradores

Para esse conjunto de entradas, temos como principais entradas os argumentos de 5 bits que selecionam os registradores para serem lidos ou sofrer a escrita. As outras entradas são o relógio para manter o sincronismo e sinais emitidos pelo controle que dão ou não permissão para resetar todos os registradores (iCLR) e permissão para escrever (iRegWrite).

B.2.b Verificação

Em seguida fazendo o teste do banco de registradores. Para o teste verificamos que sempre na subida do relógio ocorre a escrita, e por isso os registradores que leem o valor só atualizam o valor um pouco depois da subida do relógio. Verificamos também que a escrita só ocorre se a permissão (iRegWrite) estiver ativa e o RST funciona perfeitamente.

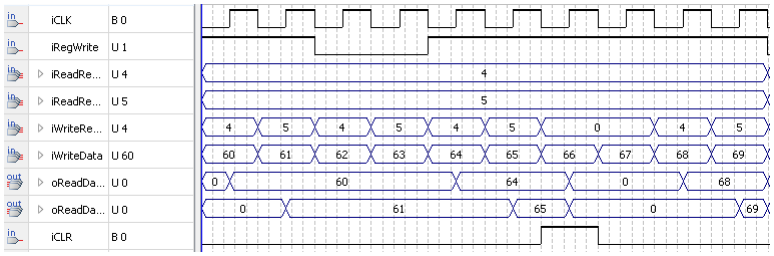

Figura 7: teste do banco de registradores

B.2.c Requisitos

Fazendo a análise na ferramenta timequest, os parâmetros necessários para o datasheet são:

- Numero de elementos lógicos: 2149
- Frequência máxima: 52.23 MHz
- Setup Time: 10.832 ns
- Hold Time: -3.5 ns
- Propagation Delay: 21.692 ns

B.3 ULA de Ponto Flutuante

B.3.a Análise de Funções

A FPU tem uma aplicação bem mais específica que a CPU, ela trabalha com operações numéricas, portanto tem menos funcionalidades. A FPU a ser trabalhada tem 12 funções, onde as operações 1-7 são operações numéricas, 8-10 são funções lógicas que apenas alteram o banco de flags e mais duas operações que convertem o tipo de dado guardado para possibilitar a comunicação com a CPU.

Sinal de Controle	Operação	oresult
1	ADD	A+B
2	SUB	A-B
3	MUL	A*B
4	DIV	A/B
5	SQRT	
6	ABS	|A|
7	NEG	-A
8	CEQ	0*
9	CLT	0*
10	CLE	0*
11	CVTSW	A single
12	CVTWS	A word

Tabela 4: Funcionalidades da ULA de ponto flutuante

B.3.b Simulação

Para a simulação em forma de onda, coloquei os valores 4 e -4 e verifiquei os cálculos. Os valores são arbitrários e com a principal função de ter como resultados valores que não interferem na mantissa dos resultados e precisar de poucos ciclos (já que tem o mesmo expoente), assim facilitando a visualização e a correção dos erros.

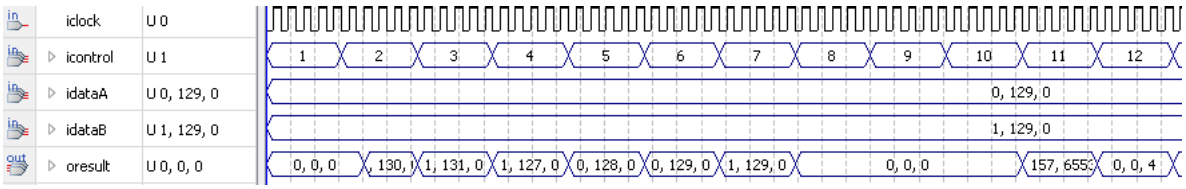

Figura 8: teste da ULA de ponto flutuante

B.3.c Sintetização da FPGA

Para o funcionamento, fizemos um experimento em verificar as operações com liberdade de alterar um operando e mantendo o outro fixo. O intuito é mostrar de forma simplificada que a FPU funciona de forma adequada sem precisar de muita complexidade no vídeo. O vídeo de teste se encontra aqui

B.3.d Requisitos

Ao fazer as simulações com o timequest, obtivemos os seguintes parâmetros de projeto:

- Numero de elementos lógicos: 3488

- Frequência máxima: 51.97 MHz
- Setup Time: 16.986 ns
- Hold Time: -0.387 ns
- Propagation Delay: 10.372 ns

B.4 Banco de Registradores do Coprocessador 1

B.4.a Construção

A banco de registradores da FPU é idêntico ao da CPU. A mudança da forma de se escrever os dados (da notação em inteira em complemento de dois para a notação em F.P.) e a ULA mais complicada não altera nada do ponto de vista dos registradores.

B.4.b Verificação

Ele é idêntico ao banco da CPU, portanto o teste que foi aplicado lá funciona igualmente aqui. O banco funciona perfeitamente bem para um processador uniciclo e para um processador multiciclo, bastando o controle definir bem em que momento a permissão de escrita pode ser dada.

B.4.c Requisitos

Ao fazer a análise pelo timequest, observamos pequenas diferenças em relação ao teste da ULA. A razão é que a implementação do banco de registradores não foi escrita exatamente com o mesmo arquivo e por isso os resultados acabam sendo levemente diferentes um do outro.

- Numero de elementos lógicos: 2096
- Frequência máxima: 51.97 MHz
- Setup Time: 12.859 ns
- Hold Time: -3.487 ns
- Propagation Delay: 22.044 ns

Referências

[1] Patterson, D.A.; Hennessy, J.L. *Computer Organization and Design* 5th ed. Elsevier, 2014.

[2] Rabinowitz, S.; Wagon, S. *A Spigot Algorithm for the Digits of Pi.* Disponível em: `http://www.mathpropress.com/stan/bibliography/spigot.pdf`. Acesso em 3 de Maio de 2014.