Simulador de Arquiteturas Pipeline Superescalares

Leonel Pablo Tedesco[1], Xano Trevisan Kothe[1]

[1]Engenharia de Computação – Universidade de Santa Cruz do Sul (UNISC)
Av. Independência, 2293 – Santa Cruz do Sul – RS – Brasil

leoneltedesco@unisc.br, xanokothe@gmail.com

Abstract. *This work has focus the development of a simulator for superscalar architectures. In this simulator, one can set execution parameters and add new specific functional units. As a case study, it is presented the performance obtained by inserting a string handling unit with superscalar basic configuration. With the addition of this unit, it has achieved a gain of up to 2 times compared to an unmodified architecture.*

Resumo. *Este trabalho possui como foco o desenvolvimento de um simulador de arquiteturas superescalares. Neste simulador, pode-se configurar parâmetros de execução e acrescentar novas unidades funcionais específicas. Como estudo de caso, é apresentado o desempenho obtido pela inserção de uma unidade de manipulação de strings a uma configuração superescalar básica. Com o acréscimo desta unidade, atingiu-se um ganho de até 2 vezes em comparação a uma arquitetura não modificada.*

1. Introdução

As arquiteturas superescalares constituem a base de construção dos processadores modernos. Tais arquiteturas possuem suporte de diversas unidades funcionais para a execução dos mais variados tipos de instruções. Devido a esta diversidade, muitos fatores podem influenciar o fluxo de execução de um conjunto de instruções, entre eles: o grau da máquina, capacidade de prever desvios, número de unidades funcionais e seus respectivos tempos de execução.

Os processadores evoluiram de uma arquitetura monociclo (todas as instruções demoram um ciclo), onde o período de relógio do processador é determinado pela instrução mais longa, para uma arquitetura *pipeline*, que dividem as tarefas em estágios e unidades funcionais heterogêneas que podem executar instruções de forma paralela. Com isso, o tempo de um ciclo pode ser reduzido, aumentando a performance [1,2,3].

A Figura 1 mostra a diferença estrutural de uma arquitetura monociclo para uma *pipeline*, sendo que nesta última, diferentes instruções podem estar em diferentes estágios, que são suportados por *buffers* intermediários.

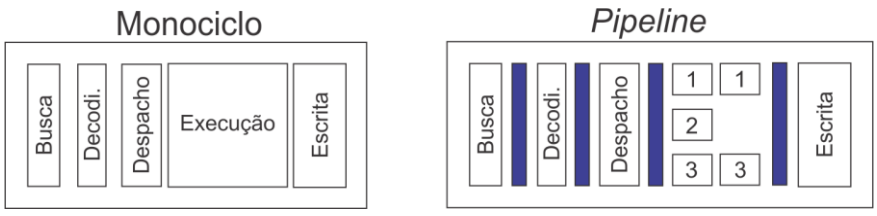

Figura 1. Monociclo vs *Pipeline*

Porém, a utilização de diversas unidades funcionais com *pipeline* trouxe uma nova gama de problemas inexistentes em arquiteturas monociclo, que entre elas são as dependências, que podem ser classificadas em:

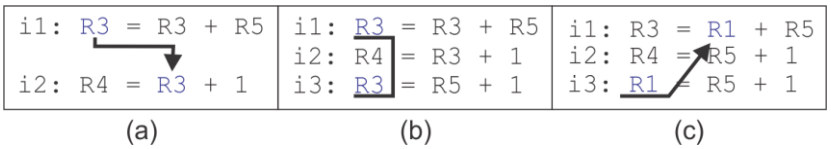

Figura 2. Dependências de dados

- Dependência verdadeira (Figura 2.a), que acontece quando um dos operandos de uma instrução dependem dos valores de uma instrução anterior;
- Dependência de saída (Figura 2.b), que acontece quando instruções escrevem em um mesmo registrador;
- Antidependência (Figura 2.c), que acontece quando uma instrução escreve em um registrador, e uma instrução anterior tem esse registrador como operando;
- Dependência de desvio, que acontece quando uma instrução precisa alterar o contador de programa, ou seja, desviar o fluxo de execução para outra parte do código;
- Conflito de recursos, que acontece principalmente quando instruções precisam utilizar a mesma unidade funcional, cabendo ao escalonador despachar uma instrução de cada vez;

Dependências de saída e antidependências ocorrem nas arquiteturas superescalares que executam instruções fora de ordem, por mais que a arquitetura tire proveito da execução paralela, o comportamento final deve ser igual a uma execução sequêncial, isso é chamado de "iniciação fora de ordem com terminação em ordem". O processador deve ser capaz de detectar dependências, e despachar as instruções para as unidades funcionais ou armazena-las em buffers até que o conflito seja resolvido [1,2,3].

Em relação à execução das instruções, devido à diversidade da modelagem do *pipeline* dos processadores superescalares, é conveniente adotar que instruções de comparação, como *beq* (*Branch if Equal*), necessitam menos ciclos em relação a instruções que acessam a memória, como *load* (que carrega um valor da memória para um registrador). A execução de instruções de comparação é feita utilizando uma lógica combinacional, que não utiliza *buffers* intermediários ou memórias, e por isso, em hardware, é de baixo custo computacional [2].

Instruções de *load* utilizam mais ciclos porque acessam duas vezes o banco de registradores (para ler e escrever) e ainda podem acessam níveis diferentes de memória (como memória *cache,* memória RAM e disco rígido). Essa busca tem um custo computacional, que pode ser alto ou não, dependendo da modelagem do processador. Atualmente, processadores contam com vários níveis de memória *cache*, como L1, L2 e L3 a fim de diminuir o tempo de busca dos valores na memória [2].

Em contrapartida, instruções aritméticas, como *add* (soma de registradores), necessitam de um número intermediário de ciclos de execução, pois utilizam lógica combinacional e acesso ao banco de registradores [2].

2. Simulador proposto

Este trabalho possui como foco o desenvolvimento de um simulador didático de arquiteturas superescalares, com a vantagem de as unidades funcionais serem modulares, sendo possível medir o comportamento e desempenho das aplicações executadas com diversas configurações. Alguns exemplos de simuladores são: "MARS" (*MIPS Assembler and Runtime Simulator*) [4], "ProcessorSim MIPS" [5] e "SimpleScalar" [6].

O simulador "MARS" [4] não ilustra o caminho de dados, sendo muito útil na visualização de resultados finais de simulação na tela, como o estado da memória, banco de registradores e contador de programa. Já o "ProcSim" [5] é uma ferramenta gráfica, que ilustra bem os caminhos de dados para várias classes de instruções. Por sua vez o "SimpleScalar" [6] simula um processador pipeline, além de possibilitar o acréscimo de novas instruções. A maioria das ferramentas existentes simulam o comportamento de processadores monociclo, enquanto o simulador desenvolvido, simula o comportamento de um *pipeline* superescalar de forma visual.

Uma vez executado um dado conjunto de instruções, é possível visualizar a utilização das unidades funcionais em cada ciclo, e ainda, medir seu desempenho utilizando uma métrica padrão.

O esquemático de alto nível do simulador é mostrado na Figura 3, e contém o módulo "Processador" que comunica-se com os módulos "Instruções" e "Memória". Esse tipo de arquitetura se chama *Harvard*, e tem como característica principal a separação entre uma memória de dados e uma memória de instruções. A ferramenta apresenta a nível visual o detalhe do módulo "Processador". Na Figura 4, é ilustrado o comportamento da execução de um conjunto de instruções.

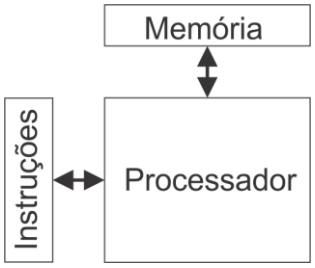

Figura 3. Diagrama de alto nível

Com essas informações, é possível determinar os recursos utilizados (por exemplo, quais unidades serão mais utilizadas para determinado código), detectar gargalos causados pelas dependências de dados e gargalos nas unidades funcionais, além disso, analisar os *buffers* de despacho e de terminação.

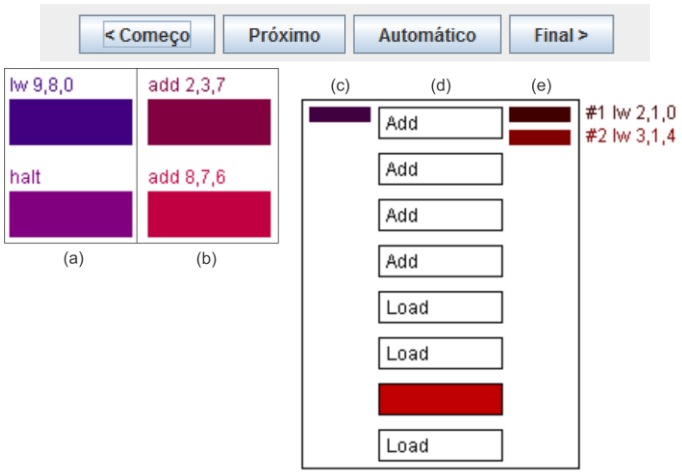

Figura 4. Simulador de *pipeline* superescalar de grau 2 após 4 ciclos

Os diferentes estágios de uma arquitetura *pipeline* superescalar no simulador são representadas na Figura 4 como:

(a) Busca de instruções com duas unidades paralelas;

(b) Decodificação de instruções com duas unidades paralelas;

(c) *Buffer* de despacho de instruções;

(d) Unidades funcionais e suas respectivas cargas de instruções;

(e) *Buffer* de terminação de instruções;

Uma característica adicional deste simulador é a possibilidade de construir unidades funcionais (como somadores de inteiros ou de ponto flutuante, unidades de comparação, operações com a memória de dados), e atribuir tempos de relógio e comportamentos diferenciados, assim sendo possível descrever um processador básico, que, por exemplo, simule o comportamento de um sistema embarcado de propósito específico. Além disso, este processador oferece suporte de previsão de desvios de 2 *bits*.

3. Unidade funcional "*strlen*"

Blocos de instruções de manipulação de *strings* estão presentes em diversas aplicações. Um exemplo são compiladores e interpretadores (tanto na construção quanto na utilização) que, basicamente, empregam analisadores léxicos (*scanners*). E em nível de usuário, podem encontrar o processamento de *strings* em navegadores de internet (na descrição de interfaces web e interpretadores *javascript*) e processadores de texto.

Uma das funções de *string* mais utilizadas é a *strlen*, pode ser encontrada na biblioteca padrão para tratamento de *strings* da linguagem de alto-nível C e C++, "<string.h>". Essa função recebe como parâmetro o endereço da memória onde inicia a *string*, e retorna um inteiro que é o tamanho desta *string*.

O processamento desta função é essencialmente sequencial, onde um laço de repetição varre uma *string* até um endereço final em busca de um caracter zero (*null*). Desta forma o tamanho é calculado pela diferença entre o endereço final e o endereço

inicial da *string*. A Figura 5 mostra como esse algoritmo pode ser escrito na linguagem C.

```
size_t strlen(const char *str)
{
        const char *s;

        for (s = str; *s; ++s)
                ;
        return (s - str);
}
```

Figura 5. Função strlen para o sistema bsd [7]

No contexto deste trabalho, foi criada uma unidade funcional especial para processamento de *strings*. Tal unidade é escalonada utilizando uma instrução proposta, chamada de "*strlen*", que possui como parâmetros:

- Um registrador destino, que terá na terminação da instrução, como conteúdo, o tamanho da *string*;
- E um registrador origem, que contém o endereço inicial da *string*;

A Figura 6 apresenta um esquemático simplificado para a unidade funcional proposta (*strlen*). A entrada (Figura 6.a) será o endereço inicial da *string*, e depois exercerá a função cursor (para varrer a memória entrada após entrada). Este cursor é utilizado como base para a busca do caracter na memória. Se o caracter buscado não for zero, o endereço é incrementado. Porém se o caracter for zero, a execução termina, e o tamanho da *string* é devolvido (Figura 6.b).

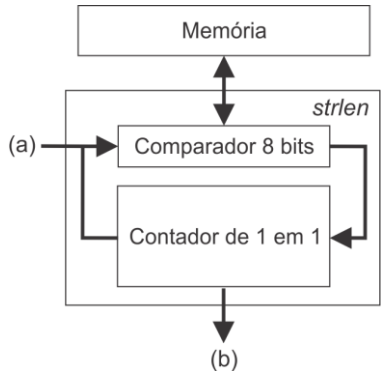

Figura 6. Esquemático do módulo *strlen*

4.Resultados obtidos

Esta seção apresenta os resultados de execução do simulador proposto, e tem como objetivo validar a arquitetura implementada e avaliar o desempenho obtido com o acréscimo de um módulo específico. Como parâmetros de execução da simulação, o ambiente de teste foi configurado com:

- 4 unidades funcionais de soma de inteiros (que executam instruções de "ADD" e "ADDI");
- 4 unidades funcionais de *load* de 1 byte (que executam instruções de "LB");

- 4 unidades funcionais de *Branch if Equal* (que executam instruções de "BEQ");
- 1 unidade funcional de *strlen* (que executa a instrução proposta "*STRLEN*");

Os ciclos de relógio para as unidades funcionais foram adotados segundo a Tabela 1.

Tabela 1. Número de ciclos por unidade funcional

Nome da unidade funcional	Número de ciclos
Soma de inteiros	2
Load de 1 byte	4
Branch if Equal	1
Strlen (a)	5n

Para a unidade *strlen* (Tabela 1.a), o número de ciclos depende do tamanho da *string* de entrada (*n*), sendo que são necessários cinco ciclos por caracter para sua terminação. Esse valor foi adotado pois no projeto pretende-se, nos primeiros quatro ciclos, buscar o caracter na memória e compará-lo, e no ciclo seguinte somar ou não o endereço atual. Também foi adotado que a soma de inteiros precisa de dois ciclos para ser completada, enquanto a instrução de *load* (acesso a memória) quatro ciclos, e salto condicional *Branch if Equal* (compara se os registradores são iguais) como um ciclo.

Essa unidade funcional tem um comportamento semelhante ao código apresentado na Figura 5. A diferença é que toda a lógica do laço de repetição estaria dentro da unidade funcional. Para testar diferentes casos, é alterado o tamanho da *string* de entrada (Tabela 2.a).

Tabela 2. Número de ciclos sem (b) e com (c) a instrução específica

Tamanho da *string* (a)	Função *strlen* (b)	Unidade Funcional Específica (c)	(b) / (c)
2	34	19	1,8
4	54	29	1,9
8	94	49	1,9
16	174	89	2,0
32	334	169	2,0
64	654	329	2,0
128	1294	649	2,0
256	2574	1289	2,0

O número de ciclos necessários para que uma função *strlen* genérica termine, é representado pela coluna "b" da Tabela 2. Enquanto a coluna "c" contém o número de ciclos para a unidade funcional proposta.

A relação entre "função *strlen*" (Tabela 2.b) e "unidade funcional específica" (Tabela 2.c), mostra um ganho de performance em todos os casos testados, necessitando aproximadamente metade dos ciclos de execução para se obter o mesmo resultado. Outro dado importante é o fato da "função *strlen*" precisar de três instruções diferentes em comparação com apenas uma da unidade funcional específica. Consequentemente,

mais unidades estão ativas e potencialmente consumindo mais recursos (energia e unidades funcionais).

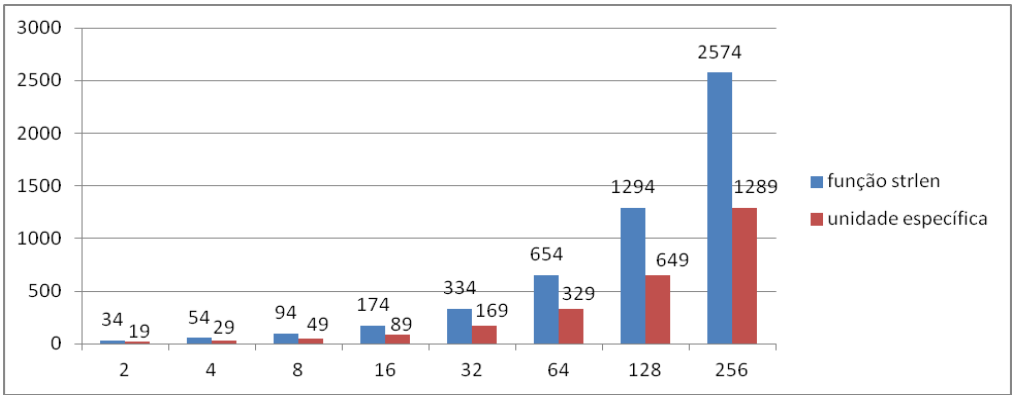

Figura 7. Gráfico da Tabela 1

Com o simulador, foi possível rapidamente perceber que o problema da abordagem seqüencial da função, não poderia ser resolvido aumentando o número de unidades funcionais, ou o número de unidades de busca e decodificação ou ainda aumentando o tamanho dos *buffers*. O problema está na dependência de dados.

Mesmo que a arquitetura tenha a capacidade de prever desvios com eficiência (falhando apenas na última interação do laço), o registrador principal ("const char *s" na Figura 5) tem uma dependência de dados com ele mesmo ("++s") que deve ser resolvida a cada interação, criando um impedimento na execução paralela das instruções.

4. Conclusão

O simulador mostrou-se uma ferramenta dinâmica e muito útil para o estudo do caso proposto (processamento de *strings*). Devido a sua modularização, foi criado e integrado ao processador um modulo *strlen*.

Os resultados obtidos mostram que é viável desenvolver um módulo de processamento de *strings*, de modo a melhorar o desempenho computacional geral das aplicações que rodam em processadores superescalares. Estes resultados também serão utilizados como motivação para desenvolver mais pesquisas, trabalhos e implementações sobre o processamento de *strings* em hardware, além de ser um recurso de apoio em aulas direcionadas para Arquitetura de Computadores.

Referências

[1] Hennessy, J.L., Patterson, D.A. (1996) "Computer Architecture: A Quantitative Approach", Morgan Kaufmann.

[2] Patterson, D.A. e Hennessy, J.L. (1998) "Computer Organization and Design: The Hardware/Software Interface", Morgan Kaufmann.

[3] Stallings, W. (2002) "Arquitetura e Organização de Computadores", Prentice-Hall.

[4] "MARS MIPS simulator - Missouri State University " http://courses.missouristate.edu/KenVollmar/MARS/

[5] “A Visual MIPS R2000 Processor Simulator – Freeware” http://jamesgart.com/procsim/

[6] “SimpleScalar LLC”, http://www.simplescalar.com/

[7] “openbsd”, http://www.openbsd.org/cgi-bin/cvsweb/src/lib/libc/string/strlen.c?rev=1.7;content-type=text%2Fplain;only_with_tag=MAIN