Algoritmo de Tomasulo

Cristiano D. Ferreira RA.: 049239

ABSTRACT

Nos dias atuais os processadores estão cada vez mais rápidos. A utilização de *pipeline* explorando o paralelismo no nível de instrução visa reduzir o CPI médio do processador e com isso melhorar seu desempenho. Porém, com essa abordagem surgem também problemas como os conflitos de dados e que precisam ser tratados. O objetivo desse trabalho é apresentar o algoritmo de Tomasulo, uma técnica de escalonamento dinâmico de instruções que utiliza renomeação e "buferização" de registradores com o intuito de gerenciar esses conflitos de dados e extrair um melhor desempenho dos processadores.

Keywords

Algoritmo de Tomasulo, pipeline, conflitos de dados

1. INTRODUÇÃO

A melhora no desempenho é uma meta muito visada no desenvolvimento de processadores. *Pipeline* [4] e paralelismo no nível de instruções *(ILP - instruction-level parallelism)* [5] buscam melhorar esse desempenho através da execução de instruções em paralelo. Por exemplo, considere uma unidade de ponto flutuante em que a soma e a multiplicação sejam realizadas em unidades distintas. Dessa forma, no seguinte trecho de código

```
MULT.D   F3, F5, F4
ADD.D    F0, F1, F2
```

a operação de adição e a de multiplicação podem executar em paralelo. Com isso, processadores que utilizam essa técnica, como os *superescalares* [6], chegam a possuir *CPI (Cycles per Instruction)* menores que 1.

Nessa abordagem o resultado da execução em paralelo das instruções deve ser o mesmo obtido se a execução fosse seqüencial. Ou seja, no exemplo anterior, considere que a latência da multiplicação em ponto flutuante seja 10 ciclos e a da soma 2 ciclos. Ao final da execução em paralelo das instruções, o valor armazenado em F3 deve ser o produto entre os valores de F4 e F5, e o valor armazenado em F0 deve ser a soma entre F1 e F2, mesmo que a operação de adição termine sua execução primeiro.

Porém há situações em que se devem tomar certos cuidados para que essa propriedade seja garantida. Observe o trecho de código:

```
MULT.D   F3, F5, F4
ADD.D    F3, F1, F2
```

Nesse exemplo, como a operação de multiplicação é executada em 10 ciclos, se não for dado nenhum tratamento especial, ao final da execução desse trecho de código o valor armazenado em F3 é o produto entre F4 e F5, e não a soma entre F1 e F2. Esse problema caracteriza um *conflito de dados (data hazard)* [3, 1].

Para explorar o IPL e também gerenciar os conflitos de dados duas abordagens são geralmente utilizadas, o *escalonamento estático* [2] e o *escalonamento dinâmico* [1]. Na primeira abordagem, o escalonamento das instruções é realizado em nível de *software*, pelo compilador. Na segunda, o escalonamento é feito em nível de *hardware*. Uma desvantagem do escalonamento dinâmico é o fato de aumentar significativamente a complexidade do processador. Como vantagens, essa abordagem permite tratar casos em que as dependências não são conhecidas em tempo de compilação (por exemplo, referências a endereços de memória) e permite que um código compilado tendo em mente determinados parâmetros do *pipeline* possa ser executado com a mesma eficiência quando esses parâmetros são alterados.

Este trabalho tem como objetivo mostrar o algoritmo de Tomasulo[7]. Essa técnica foi desenvolvida na implementação da unidade de ponto flutuante do IBM 360/91 e tem o objetivo de realizar o escalonamento dinâmico de instruções minimizando os conflitos de dados. Vários processadores modernos implementam o algoritmo de Tomasulo, dentre eles, Pentium III/4, MIPS R10000/R12000, AMD K5/K6/Athlon, Alpha 21264[1] e IBM Power2.

Na seção 2 os conflitos de dados serão caracterizados. Na seção 3 o algoritmo de Tomasulo será mostrado. E na seção 4 serão apresentadas as conclusões desse trabalho.

2. CONFLITOS DE DADOS

Como dito anteriormente, o resultado final da execução de instruções em paralelo deve ser o mesmo resultado da

[1]Esses processadores implementam o algoritmo de Tomasulo com *especulação*, o que foge ao escopo deste trabalho

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Segundo Trabalho de MO401 2005 UNICAMP, Campinas BRA
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

execução seqüencial. Mas, conflitos de dados podem afetar processamento de um programa e gerar resultados indesejados. Por isso, esses conflitos devem ser detectados e tratados para evitar que influam na execução de um determinado programa.

Os conflitos de dados podem ser caracterizados em três tipos, dependendo da ordem de leitura e escrita das instruções. A ordem no programa que deve ser preservada pelo *pipeline* define os nomes dos conflitos. Sejam duas instruções i e j, tal que i ocorre antes de j na execução serial do programa. Assim, os conflitos de dados são:

- *RAW (read after write)* – j lê o valor de um determinado registrador em que i irá gravar um resultado, porém, ainda não o fez. Neste caso, j lerá erroneamente o valor antigo desse registrador. O correto seria primeiro i gravar o valor nesse registrador para somente depois j ler. É o que ocorre no exemplo:

  ```
  MULT.D   F0, F1, F2
  ADD.D    F3, F4, F0
  ```

  Nesse exemplo, a instrução ADD.D (j) tem como operando o registrador F0. A instrução MULT.D (i) deve gravar um valor nesse registrador antes que ADD.D o leia. Mas como a multiplicação necessita de 10 ciclos para ser completada, ADD.D lerá o valor antigo de F0.

- *WAW (write after write)* – i escreve em um determinado registrador após j escrever nesse mesmo registrador. Com isso as escritas ocorrerão na ordem inversa à que deveria ocorrer. Ao final da execução, o registrador terá o valor escrito por i, embora o correto fosse conter o resultado escrito por j. Essa situação é ilustrada no exemplo a seguir:

  ```
  MULT.D   F3, F2, F4
  ADD.D    F3, F1, F0
  ```

  Nesse caso, F3 deveria conter o resultado da soma entre os valores de F1 e F0. Mas, como a MULT.D (i) é executado em 10 ciclos e ADD.D (j) em apenas 2, o valor a ser armazenado em F3 ao final da execução será o produto entre F2 e F4.

- *WAR (write after read)* – j escreve em um determinado registrador que é operando de i antes de i conseguir lê-lo. Com isso, i lerá incorretamente o valor escrito por j. Este problema ocorre quando temos algumas instruções que gravam resultados muito cedo e outras que lêem muito tarde os operandos. Por exemplo, observe o trecho de código a seguir:

  ```
  MULT.D   F3, F2, F4
  MULT.D   F0, F5, F6
  ADD.D    F5, F2, F4
  ```

  Considerando que há apenas uma unidade para a multiplicação de ponto-flutuante, e uma para a adição, a segunda instrução MULT.D (i) só poderá ser executada após a execução da primeira. Mas ADD.D (j) termina sua execução antes disso e escreve o valor da soma no registrador F5. Assim, quando MULT.D F0, F5, F6 for executada, o valor contido no registrador F5 já será o valor atualizado por ADD.D.

3. ALGORITMO DE TOMASULO

O algoritmo de Tomasulo é um esquema que permite a execução de instruções em paralelo através do gerenciamento dos conflitos de dados que podem ocorrer. Desenvolvido na unidade de ponto flutuante IBM 360/91 esse esquema introduz a *renomeação de registradores*, com o objetivo de minimizar os conflitos WAW e WAR, e verifica a disponibilidade de operandos para evitar conflitos RAW.

A implementação do algoritmo de Tomasulo mostrada neste trabalho se concentra na unidade de ponto-flutuante e na unidade de carga e armazenamento no contexto do conjunto de instruções MIPS. A arquitetura do *hardware* necessário para a implementação do algoritmo é apresentada na seção 3.1. A seção 3.2 define a renomeação de registradores. Finalmente, os estágios da execução de uma instrução na arquitetura apresentada de acordo com o algoritmo de Tomasulo são mostrados na seção 3.3.

3.1 Arquitetura do hardware

A arquitetura da unidade MIPS de ponto flutuante utilizada na implementação do algoritmo de Tomasulo é mostrada na figura 1, extraída de [1]. Nessa figura, podemos observar que as instruções enviadas a partir da unidade de memória são inseridas em uma fila. Essas instruções, emitidas em ordem FIFO, são enviadas para as *estações de reserva, buffers de carga* e *buffers de armazenamento*. Depois, as instruções são executadas nas unidades correspondentes (*unidade de memória, somadores de ponto flutuante* e *multiplicadores de ponto flutuante*). Os resultados são colocados no *barramento de dados comum (CDB – common data bus)*. Isso permite que os resultados sejam repassados diretamente para o *banco de registradores*, as estações de reserva e os *buffers* de armazenamento.

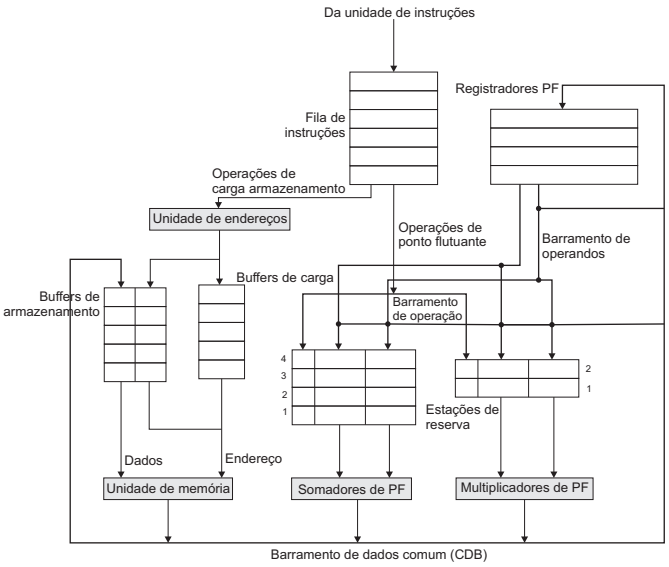

Figure 1: Arquitetura do *hardware*.

Cada estação de reserva é atrelada a uma determinada unidade funcional. As estações contêm instruções que esperam pela disponibilização da unidade funcional e pelos operandos necessários para sua execução. Cada uma dessas estações armazena um conjunto de campos utilizados para

detectar e resolver conflitos. Cada estação de reserva é composta pelos seguintes campos:

- Op – Operação a ser executada sobre os operandos $S1$ e $S2$;
- Qj e Qk – As estações de reserva que produzirão o operando de origem. O valor zero indica que o respectivo operando de origem já está disponível em Vj ou Vk, ou ele não é necessário;
- Vj e Vk – O valor dos operandos de origem. Em instruções de carga, o campo Vj é utilizado para conter o campo de deslocamento;
- A – Utilizado para armazenar informações correspondentes ao cálculo de endereços de memória para carga ou armazenamento. Inicialmente, esse campo contém o imediato da instrução. Quando disponível, o endereço efetivo (soma entre o imediato e o registrador base) é armazenado nesse campo;
- $Busy$ – Indica que a estação de reserva e a unidade funcional que a acompanha estão ocupadas;

Os *buffers* de carga e os de armazenamento contêm os componentes necessários para o cálculo do endereço de memória efetivo e se comportam de modo semelhante às estações de reserva. Esses *bufers* também possuem um campo A que conterá o endereço efetivo calculado. Os *buffers* de carga armazenam as instruções de carga pendentes que aguardam pelos dados da memória e os valores trazidos da memória que esperam pela disponibilização do CDB. Já os *buffers* de armazenamento contêm instruções de escrita que aguardam pelos valores a serem armazenados na memória e também o endereço e o valor a ser escrito, até que a unidade de memória esteja disponível.

Os somadores de ponto flutuante realizam a adição e subtração e os multiplicadores de ponto flutuante implementam a multiplicação e a divisão.

O banco de registradores possui um campo, Qi, que indica, para cada registrador r, a estação de reserva que contém a operação cujo resultado deve ser armazenado em r. Se o valor de Qi for zero, nenhuma instrução que possui o r como destino está sendo executada. Assim, o valor de r é o valor já nele contido.

3.2 Renomeação de registradores

Os conflitos WAW e WAR podem resolvidos através da *renomeação de registradores*. Essa técnica utiliza registradores temporários para renomear todos os registradores de destino que aparecem em instruções anteriores tanto como origem quanto como destino. Se um dado registrador R é renomeado para T, por exemplo, em quaisquer usos subseqüentes de R como origem, esse registrador será renomeado para T. Por exemplo, considere o seguinte trecho de código:

```
1  MULT.D  F1, F4, F5
2  ADD.D   F1, F2, F3
3  MULT.D  F6, F7, F2
4  ADD.D   F7, F2, F4
5  ADD.D   F8, F1, F4
```

Nesse exemplo, há dois conflitos de dados possíveis. A utilização do registrador $F1$ como destino das instruções 1 e 2 permite o surgimento de um conflito WAW. E ainda há a possibilidade da existência de um conflito WAR entre as instruções 3 e 4. Para resolver esses problemas, a renomeação de registradores pode ser aplicada nesse trecho de código utilizando dois registradores temporários $R1$ e $R2$. Dessa forma, a seqüência de código pode ser reescrita como segue:

```
1  MULT.D  F1, F4, F5        MULT.D  F1, F4, F5
2  ADD.D   F1, F2, F3   ⇒    ADD.D   R1, F2, F3
3  MULT.D  F6, F7, F2        MULT.D  F6, F7, F2
4  ADD.D   F7, F2, F4   ⇒    ADD.D   R2, F2, F4
5  ADD.D   F8, F1, F4   ⇒    ADD.D   F8, R1, F4
```

Como pode ser observado, a renomeação de registradores eliminou os conflitos WAW e WAR existentes no código original.

No algoritmo de Tomasulo, as estações de reserva fazem o papel dos registradores temporários e realizam a renomeação. Isso ocorre porque as instruções são emitidas na ordem de execução do programa. Assim, quando uma instrução i é emitida, se não houver um conflito RAW, os operandos são copiados do banco de registradores para a estação de reserva. Se houver esse conflito, i receberá o operando pendente logo que estiver disponível, diretamente do CDB. Dessa forma, não é possível que uma instrução emitida após i sobrescreva os operandos de i e assim, o conflito WAR é eliminado. E ainda, se duas ou mais instruções tentam gravar em um determinado registrador, apenas a última consegue completar essa tarefa. Isso porque, para cada registrador, é mantido um campo Q_i, como mostrado na seção 3.1. E nesse campo é armazenado o número da estação de reserva contém a última instrução emitida que irá gravar no registrador. Assim, elimina-se o conflito WAW.

3.3 Estágios de execução

Na arquitetura apresentada, uma instrução a ser executada percorre três estágios. Basicamente, esses estágios são:

- *Emitir* – Se houver uma estação de reserva disponível, a primeira instrução da fila de instruções é inserida nessa estação. Se os operandos estiverem nos registradores, eles são lidos, caso contrário a instrução aguarda a disponibilização dos operandos. Se não houver estação disponível, a instrução será desdobrada. Essa etapa renomeia registradores e assim, os conflitos WAW e WAR são eliminados;
- *Execução* – Se operandos estiverem disponíveis então a instrução é executada. Senão observa-se o CDB até que os operandos estejam prontos e a instrução é executada. Nesta fase os conflitos RAW são verificados;
- *Gravar resultado* – Escreve-se o resultado no CDB e a estação de reserva é marcada como disponível.

A seguir, os passos do algoritmo de Tomasulo em cada uma dessas etapas serão detalhados. Para isso, sejam rd o registrador de destino da instrução, rs e rt is registradores de origem, imm o campo imediato com extensão de sinal e r a estação de reserva ou *buffer* em que a instrução é atribuída. Sejam também RS a estrutura de dados da estação de reserva/ *buffer*, $RegisterStat$ a estrutura de dados de *status* do registrador, $Regs$ o banco de registradores e $result$ o valor escrito no CDB.

Operação de PF	**if** RegisterStat[rs].Qi ≠ 0 {RS[r].Qj ← RegisterStat[rs].Qi} **else** {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; **if** RegisterStat[rt].Qi ≠ 0 {RS[r].Qk ← RegisterStat[rt].Qi} **else** {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0}; RS[r].Busy ← yes; RegisterStat[rd].Qi ← r;
Load ou Store	**if** RegisterStat[rs].Qi ≠ 0 {RS[r].Qj ← RegisterStat[rs].Qi} **else** {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; RS[r].A ← imm; RS[r].Busy ← yes;
Load	RegisterStat[rt].Qi ← r;
Store	**if** RegisterStat[rt].Qi ≠ 0 {RS[r].Qk ← RegisterStat[rt].Qi} **else** {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};

Figure 2: Passos do estágio Emitir.

No estágio Emitir, se a instrução for uma operação de ponto flutuante então espera-se até que haja uma estação vazia. Quando houver uma estação r disponível, a instrução é emitida e é verificado se os operandos estão no banco de registradores. Se estiverem, estão os operandos são copiados para a estação. Senão, é indicado em r as estações que determinarão os operandos. Feito isso, é indicado que a estação r e a unidade funcional que a acompanha estão ocupadas e r é definida como a estação que contém a operação cujo resultado será armazenado em *rd*.

Se a instrução for de carga ou armazenamento, espera-se até que haja um buffer r disponível. Quando houver, a instrução é emitida e verifica-se se o registrador base está disponível no banco de registradores. Se estiver, é copiado para a estação, senão, é indicado em r a estação que determinará o valor do operando. Se for uma instrução de carga, r é definida como a estação que contém a operação cujo resultado será armazenado em rt. Se for uma operação de armazenamento, é verificado se o valor a ser armazenado está disponível. Se estiver, é copiado para a estação, senão, é indicado em r a estação que determinará esse valor.

Esses passos são descritos na figura 2.

No estágio Executar, enquanto os operandos não estiverem disponíveis, o CDB é monitorado a espera desses operandos para inseri-los na estação de reserva correspondente. Essa espera é realizada com o intuito de evitar conflitos RAW. Quando todos os operandos estiverem disponíveis e a unidade a ser utilizada estiver livre, se a instrução for uma operação de ponto flutuante então ela é executada. Se a instrução for de carga ou de armazenamento, a instrução não é executada até que r seja o primeiro *buffer* da fila. Quando for, o endereço efetivo é calculado. Se a instrução for de carga, é realizado o acesso à memória para leitura. A figura 3 resume esses passos.

No estágio Gravar resultado, se a instrução for uma operação em ponto flutuante ou de carga, quando o resultado está

Operação de FP	Quando (RS[r].Qj ≠ 0) e (RS[r].Qk ≠ 0) {Executar operação};
Load ou Store	Quando (RS[r].Qj ≠ 0) e r for o primeiro da fila de carga-armazenamento {RS[r].A ← RS[r].Vj + RS[r].A;}
Load	Ler Mem[RS[r].A]

Figure 3: Passos do estágio Executar.

Operação de FP ou Load	Quando execução concluída em r e CDB disponível
	∀(**if** (RegisterStat[x].Qi = r) {
	Regs[x] ← result;
	RegisterStat[x].Qi ← 0});
	∀(**if** (RS[x].Qj = r) {
	RS[x].Vj ← result;
	(RS[x].Qj ← 0});
	∀(**if** (RS[x].Qk = r) {
	RS[x].Vk ← result;
	(RS[x].Qk ← 0});
	RS[r].Busy ← no;

Store	Quando execução concluída em r e RS[r].Qk = 0 Mem[RS[r].A] ← RS[r].Vk; RS[r].Busy ← no;

Figure 4: Passos do estágio Gravar resultado

disponível, ele é gravado no CDB. Assim, esse resultado é distribuído para o registrador, as estações de reserva e os *bufers* de armazenamento que aguardam por ele. Se a instrução for de armazenamento, se o valor a ser armazenado e o endereço de armazenamento estiverem disponíveis, então é realizada uma escrita na memória. A figura 4 exibe esses passos.

Para que o programa execute corretamente, uma determinada instrução não pode executar até que todos os desvios posteriores a ela na ordem de programa sejam resolvidos.

4. CONCLUSÕES

O esquema de Tomasulo combina a renomeação de registradores e a "buferização" dos operandos de origem com as estações de reserva. Isso permite a eliminação de paradas devido a conflitos WAW e WAR. Os conflitos RAW são evitados com o atraso da execução de instruções que não estejam com seus operandos disponíveis.

Uma outra vantagem desse esquema é a distribuição da lógica de detecção de conflitos e do controle de execução, pois quando os operandos se tornam disponíveis, múltiplas instruções podem ser emitidos simultaneamente. Isso porque os resultados são disponibilizados para todas as estações

de reserva ao mesmo tempo, diretamente das unidades funcionais.

A maior desvantagem desta abordagem é a complexidade do algoritmo de Tomasulo que requer uma grande quantidade de hardware. Além disso, a performance também é limitada pela presença de um único CDB, pois ele precisa interagir com todo o *hardware* do *pipeline*.

5. REFERENCES

[1] J. L. H. e David A. Patterson. *Arquitetura de Computadores uma abordagem quantitativa*, chapter 3, pages 125–218. Editora Campus, 3 edition, 2003. Tradução da terceira edição americana.

[2] J. L. H. e David A. Patterson. *Arquitetura de Computadores uma abordagem quantitativa*, chapter 4, pages 219–282. Editora Campus, 3 edition, 2003. Tradução da terceira edição americana.

[3] G. M. Prabhu. Computer architecture tutorial. http://www.cs.iastate.edu/ prabhu/Tutorial/title.html. Acessado em 31/10/2005.

[4] C. V. Ramamoorthy and H. F. Li. Pipeline architecture. *ACM Computing Surveys (CSUR)*, 9(1):61–102, 1977.

[5] B. R. Rau and J. A. Fisher. *Instruction-level parallel processing: history, overview, and perspective*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[6] G. Smith, J.E.; Sohi. The microarchitecture of superscalar processors. In *Proceedings of the IEEE*, volume 83, pages 1609–1624, Madison, WI, USA, Dezembro 1995.

[7] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In *IBM Journal of Research and Development*, January 1967.