Arquitectura de Computadores II

3º Ano

Exercícios sobre *Pipelining*

João Luís Ferreira Sobral
Departamento do Informática
Universidade do Minho

Maio 2003

Considere o modelo de uma arquitectura MIPS com o *pipeline* da figura em anexo, sem unidade de encaminhamento de dados (*forward unit*), sem unidade de detecção de anomalias (*hazard detection unit*), com a escrita dos valores em registos (*write back*) na primeira metade do ciclo, com empates (*stall*) do *pipeline* em instruções de salto e com *cache* "quente" e de dimensão infinita. Considere agora o seguinte programa em *assembly* MIPS:

```
      mov $s1, $0
      mov $s2, $0
      mov $s0, -1024
soma: lw $t0, 0xC400($s0)
      addu $s1, $s1, $t0
      sw $s1, 0xE400($s0)
      addu $s2, $s2, $s1
      addiu $s0, $s0, +4
      bne $s0, $0, soma
      mov $s1, $0
```

1) Identifique todas as dependências de dados existentes neste programa e introduza instruções *nop* necessárias para que a execução do programa esteja correcta. Calcule o CPI mínimo e o CPI efectivo na execução deste programa neste processador. Comente os resultados.

Sugestões:

a. Analise o significado e consequência da não existência das unidades de encaminhamento e de detecção de anomalias, bem como do facto de a operação de escrita dos valores em registo ser efectuada na 1ª metade do ciclo, permitindo a leitura do conteúdo dos registos na 2ª metade; com base neste conhecimento, é possível calcular quantas instruções deverão estar entre 2 instruções com dependências críticas de dados (que no caso deste MIPS são as de RAW).

b. Analise o significado e consequência do "empate" do *pipeline* em instruções de salto, i.e., da inserção de bolhas após a execução de qualquer instrução de salto (absoluto e relativo), logo após a sua descodificação; com base neste conhecimento, é possível calcular quantas bolhas são automaticamente inseridas por salto.

c. Considere uma *cache* "quente" aquela que já contém toda a informação que o programa necessita para a sua execução (neste caso o código do programa e a área dos dados); e se considerar ainda que ela tem dimensão infinita, então tem a certeza que toda essa informação coube na *cache*, e que nenhum acesso à *cache* de instruções ou de dados será penalizado no funcionamento do *pipeline*.

d. Conte o número de instruções *x* que são precisas executar para concluir o especificado.

e. Supondo que o *pipeline* estará sempre ocupado com tarefas úteis (modelo óptimo de arquitectura), verifique quantos ciclos de relógio são precisos para completar a execução (deverá dar mais 4 que o nº de instruções, devido à latência de execução da 1ª instrução); com base nos cálculos efectuados em d) pode-se obter o CPI_{min} - que deverá ser $(x+4)/x$.

f. Identifique todas as dependências de dados entre as instruções, e veja, instrução a instrução, quantas "bolhas" são inseridas, quer através da execução de *nops* gerados pelo compilador para resolução das dependências de dados, quer ainda inseridas pela unidade de controlo de saltos no *hardware*; daqui pode contar e estimar o nº total de ciclos de relógio para executar este programa; usando como nº total de instruções a ser executadas no processador o valor *x* calculado em d. (que não é o correcto; porquê?), calcule agora o CPI efectivo.

2) Reordenando as instruções, rescreva o programa para minimizar o tempo de execução; calcule o tempo de execução do novo programa, em ciclos de relógio. Compare com o valor obtido em **1)** e comente.

Sugestão: altere a ordem de execução de instruções de modo a remover ao máximo as dependências de dados; de notar que cada instrução que escreve num registo obriga à inserção de 2 *nop* caso a instrução que se lhe segue precise de ler o conteúdo desse registo; portanto, se se conseguir alterar a ordem de execução de instruções de modo a existirem 2 instruções sem dependência de dados após uma operação que escreve num registo, a utilização do *pipeline* é melhorada; apenas com a simples reordenação de instruções deve conseguir eliminar pelo menos 3 instruções de *nop* (com um truque adicional - e que não compromete evoluções da microarquitectura - poderá ainda eliminar mais dois *nop*...).

3) Mostre o conteúdo do registo ID/EX no início do 6º ciclo de relógio.

Sugestão: identifique primeiro qual a instrução que no 5º ciclo se encontra no nível ID, sem se esquecer que na alínea anterior se conseguiu eliminar os *nops* no início do programa; agora consulte os apontamentos para construir essa instrução em binário e confirmar quais os sinais de controlo que a unidade de controlo gera nesse ciclo; o resto faz-se olhando apenas para a figura do *pipeline* em anexo.

4) Considere agora que o compilador usa a técnica de desdobramento de ciclos (*loop unroll*) (para além das melhorias introduzidas em **2)**). Use-a para processar 4 elementos de cada vez, e reescreva o código de modo a minimizar o tempo de execução. Calcule o tempo de execução do novo programa, em ciclos de relógio. Compare com o valor obtido em **2)** e comente.

Sugestão:

a. A técnica de desdobramento de ciclos serve essencialmente 2 objectivos: minimizar as penalizações resultantes das instruções de controlo de execução do ciclo e espaçar mais as instruções que obrigam à inserção de "bolhas" no *pipeline* devido a dependências de dados.

b. Identifique as penalizações no funcionamento do *pipeline* resultantes das instruções de controlo de execução do ciclo (serão só as instruções de salto?...). Reordene as instruções do novo corpo do ciclo - que processa agora 4x mais dados por cada iteração - usando, se necessário, mais registos (e verificando se dispõe do nº extra de registos que são precisos) e modificando eventualmente o próprio código do programa.

c. Use estas dicas na reescrita do código; se conseguir que cada instanciação do ciclo não tenha mais de 18 instruções úteis e 2 *nops*, então chegou a um resultado excelente!

5) Considere agora que a arquitectura possui uma unidade de encaminhamento de dados (*forward unit*) e uma unidade de detecção de anomalias (*hazard detection unit*). Modifique a solução obtida em **1)** e calcule o tempo de execução do novo programa, em ciclos de relógio. Compare com os valores obtidos em **1)** e **2)** e comente.

Sugestão: lembre-se que a introdução da unidade de encaminhamento de dados vai permitir eliminar as bolhas nas dependências de dados críticas que ocorrem após a execução de operações aritméticas/lógicas; contudo, nas operações de *load* continua a haver a necessidade de o compilador inserir um *nop*; mas, se a arquitectura possuir uma unidade de detecção de anomalias, é a própria unidade de controlo da arquitectura que se encarrega de introduzir uma bolha nesses casos de dependência crítica de dados, simplificando a tarefa de geração de código do compilador (mas não melhorando o desempenho na execução...)

6) Se o valor de CPI obtido nos programas resultantes da resolução de **4)** e de **5)** fossem iguais, que conclusões tiraria quanto ao desempenho do processador na execução deste programa? (Mesmo desempenho ou distinto, e porquê.)

Sugestão: lembre-se que a principal medida do desempenho do processador na execução deste programa é o tempo que ele necessita para o executar (em ciclos de relógio), e um bom tempo não depende apenas de uma boa "máquina" - o processador - mas também de um bom "condutor" - o código gerado pelo compilador; com base nesta informação e na fórmula que nos relaciona o tempo de execução de um programa com os parâmetros da arquitectura, pode responder à questão...

7) Considere ainda a execução atrasada da instrução de salto, i.e., a arquitectura introduz um slot de salto retardado (*branch delay slot*) (para além dos melhoramentos já introduzidos na arquitectura em alíneas anteriores). Reescreva o código de modo a minimizar o tempo de execução e calcule esse tempo, em ciclos de relógio. Compare com o valor mínimo obtido em **1)** e a melhor optimização que tinha conseguido em **5)**, e comente.

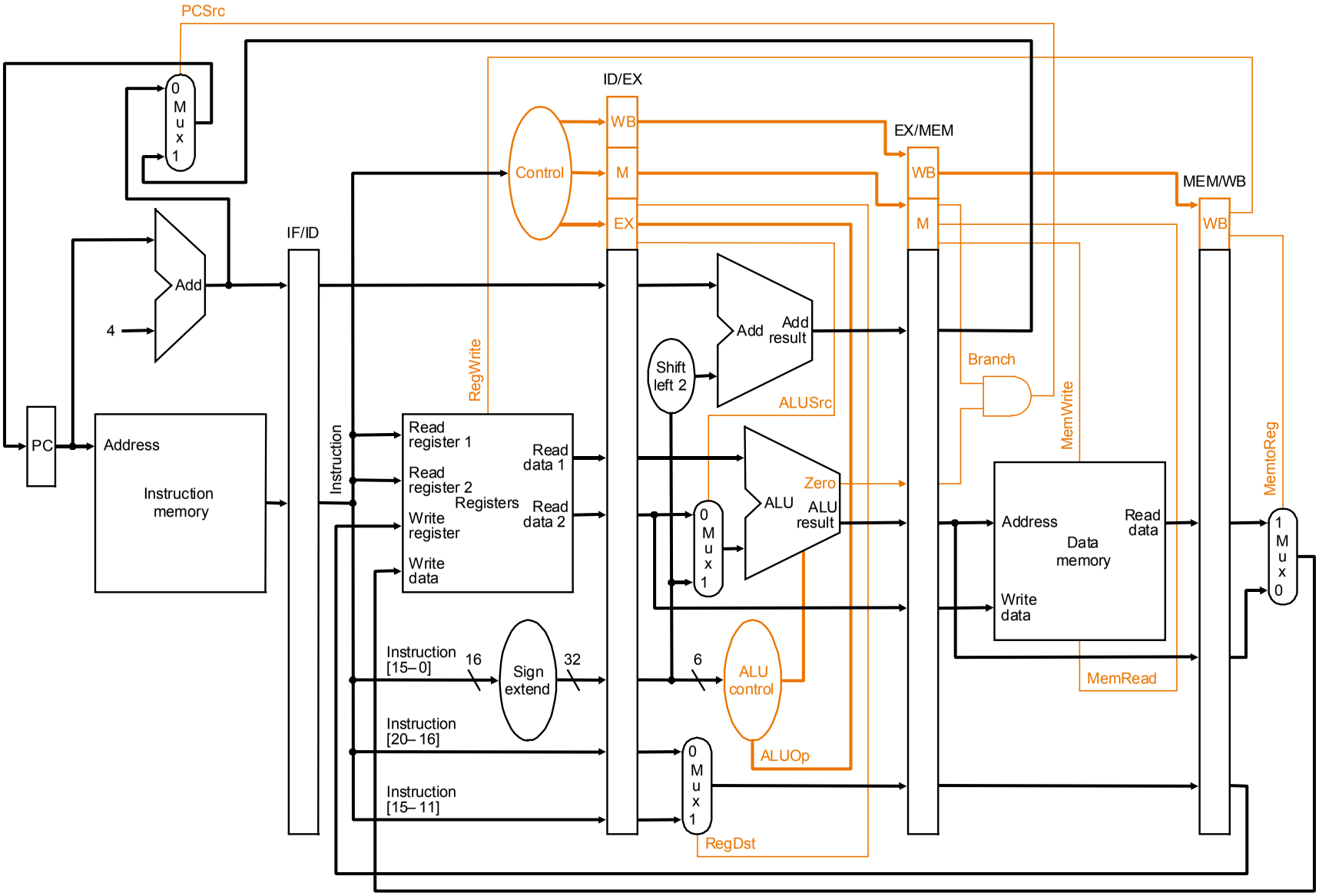
Sugestão: por execução atrasada de uma instrução - de salto ou de *load* - entende-se que uma instrução só é efectivamente executada após a instrução que a segue; no caso da instrução de salto, isto significa que, quer o salto se concretize ou não, a instrução que se encontra na palavra seguinte da memória de instruções é sempre carregada e executada; como consequência para este processador, verifica-se que, em vez da arquitectura inserir 3 bolhas após a descodificação de um *branch* (ou 1 bolha após um *jump*), a unidade de controlo deixa executar a instrução que está imediatamente a seguir e apenas introduz 2 bolhas (no caso dum *branch*, e 0 no caso dum *jump*)

8) Considere ainda uma nova optimização da arquitectura relativamente a **7)**: uma previsão estática de saltos - nunca salta. Reescreva o código de modo a minimizar o tempo de execução e calcule esse tempo, em ciclos de relógio.

Sugestão: mesmo com *branch delay slot* cada instrução de salto condicional pode ainda desperdiçar 2 instruções (bolhas) quando a previsão estática falha; mas com saltos incondicionais isto já não acontece; estas dicas servem para alguma coisa?

9) Considere agora como alternativa a **8)** uma previsão dinâmica de saltos (com 1 bit): se da vez anterior saltou, também salta agora e para o mesmo endereço, se não saltou, também não salta. Reescreva o código de modo a minimizar o tempo de execução e calcule esse tempo, em ciclos de relógio. Compare com o valor obtido em **8)** e comente.

10) Considere agora, em adição a **9)**, uma arquitectura superescalar com três unidades de execução: i) execução de acessos à memória (*load* e *store*); ii) execução de operações aritméticas e iii) execução de saltos. Qual o CPI mínimo teórico e qual o CPI efectivamente obtido na execução do programa? Comente os resultados. Qual a melhoria na execução deste programa se for introduzida uma segunda unidade para execução de operações aritméticas, passando a arquitectura a possuir quatro unidades de execução?


PCSrc
0
M
u
x
1
ID/EX
EX/MEM
MEM/WB
WB
M
EX
Control
IF/ID
Add
4
PC
Address
Instruction
memory
Instruction
RegWrite
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
Shift
left 2
Add
Add
result
ALUSrc
Branch
MemWrite
MemtoReg
ALU
ALU
result
Zero
Address
Data
memory
Read
data
Write
data
MemRead
Instruction
[15–0]
16
Sign
extend
32
6
ALU
control
ALUOp
Instruction
[20–16]
Instruction
[15–11]
RegDst