Universidade Fernando Pessoa

Arquitectura de Computadores

Ficha de Exercícios nº 6

Objectivo: resolver exercícios que permitam quantificar os ganhos de desempenho do "pipelining". O "pipelining" no MIPS.

1. Considere os tempos (em pico segundos) despendidos nas diferentes fases do "pipeline" do processador MIPS apresentados na tabela seguinte:

Instruction Class	Instruction Fetch	Register Read	ALU Operation	Data Access	Register Write	Total time	Occurrences
lw	200	100	200	200	100	800	25%
sw	200	100	200	200		700	10%
add, sub, and, or, slt	200	100	200		100	600	50%
beq	200	100	200			500	15%

   a. Identifique o tempo médio por instrução com e sem pipeline.
   b. Se o tempo gasto na ALU fosse reduzido em 25%, será que isso afetava o ganho de desempenho (*speedup*) obtido pelo "pipeline"? Se sim, em quanto? Se não, porquê?
   c. E se a operação da ALU passasse a levar mais 25% de tempo? O desempenho foi afetado?

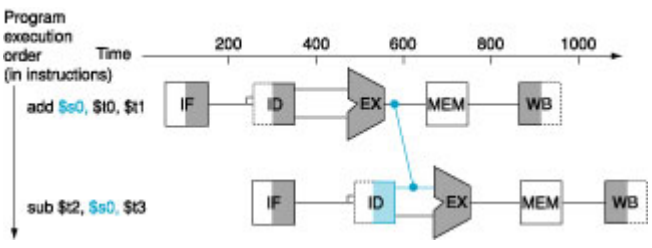
2. Um engenheiro de processadores necessita de desenhar o "pipeline" de um novo microprocessador. Ele tem um programa típico desse microprocessador com 10^6 instruções. Cada instrução necessita de 100ps para ser executada.

   a. Quanto tempo é necessário para executar este programa num processador sem "pipeline"?

   b. Considere que um processador possui um "pipeline" de cerca de 20 fases. Assumindo que esse pipeline é perfeito e que o código vai ser executado milhares de vezes qual o ganho de desempenho obtido por este processador por comparação com o da alínea anterior?

   c. Os "pipelines" reais não são perfeitos, uma vez que a sua implementação introduz algum atraso entre cada fase. Será que este atraso afeta a latência da instrução, o débito de instruções por unidade de tempo ("throughput") ou ambos? Porquê?

3. Usando um esquema semelhante ao apresentado, mostre as linhas necessárias para efectuar o "forwarding" nas 4 instruções seguintes:




```
add $3, $4, $6
sub $5, $3, $2
lw $7, 100($5)
add $8, $7, $2
```

4. Identifique todas as dependências no código seguinte. Quais são acidentes de dados que podem ser resolvidos com "forwarding"? Quais são acidentes de dados que originam a paragem do "pipeline"?

```
add $3, $4, $2
sub $5, $3, $1
lw $6, 200($3)
add $7, $3, $6
```

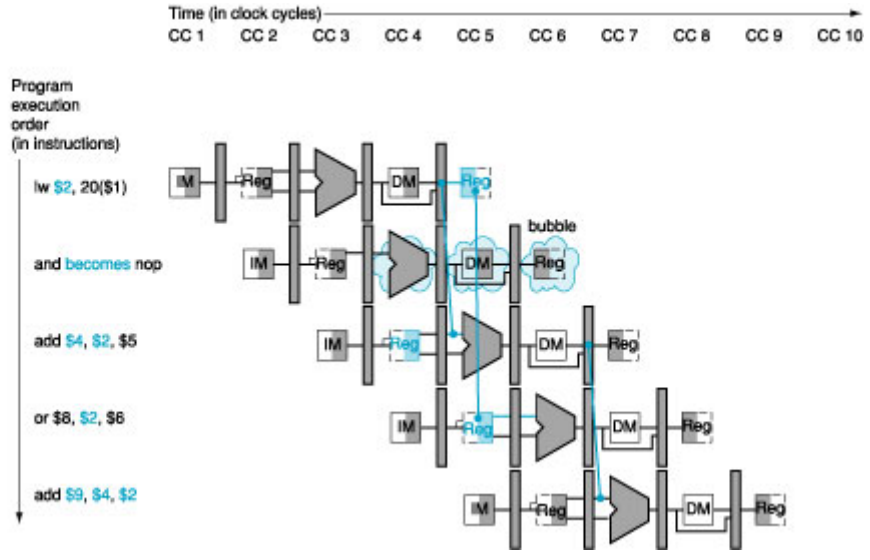
5. Reescreva o código seguinte de forma a *minimizar* o desempenho (reordenar as instruções de modo a que esta sequência leve o máximo de ciclos de relógio para ser executada mantendo o resultado da versão original):

```
Loop: lw  $2, 100($6)
      lw  $3, 200($7)
      add $4, $2, $3
      add $6, $3, $5
      sub $8, $4, $6
      lw  $7, 300($8)
      beq $7, $8, Loop
```

6. Considere o código seguinte num circuito de dados com "pipeline":

```
lw  $4, 100($2)
sub $6, $4, $3
add $2, $3, $5
```

   a. Quantos ciclos de relógio são necessários para executar este código?
   b. Mostre todas as dependências a resolver e desenhe um diagrama idêntico ao apresentado que mostre como o código é efetivamente executado (incorporando paragens do "pipeline" e "forwarding")



7. Como pode ser modificado o código seguinte de forma a usar o "delayed branch slot"?

```
Loop:  lw   $2, 100($3)
       addi $3, $3, 4
       beq  $3, $4, Loop
```

8. Considere o seguinte código em linguagem C:

```
int array[10] = {1,3,5,7,9,11,13,15,17,19};

main() {
      int i=0, s=0;
      do {
            s+=array[i];
            i++;
      } while(i<10);

      printf("valor=%d\n",s);
}
```

   a. Escreva-o em *assembly* do MIPS. Execute-o no simulador MIPS e anote o valor de saída.
   b. Supondo que vai executar este código num processador com "pipeline", rescreva-o (introduzindo a instrução **`nop`** nos locais relevantes) de forma a que possa executar corretamente na presença do "branch delay slot" e do "load delay slot".
   c. Altere a configuração do simulador MIPS para usar o "branch delay slot" e o "load delay slot" e execute o código alterado anotando o valor de saída.
   d. O código da alínea b) pode ser optimizado de forma a usar de forma produtiva o "branch delay slot" e o "load delay slot"? Se sim, efectue as alterações necessárias (justificando-as) e verifique a correção do programa executando-o novamente. Se não, justifique convenientemente.

Bibliografia:

[1] Patterson & Hennessy – *Computer Organization and Design: The hardware/software interface 4rd Ed* – MKP 2009.

University Fernando Pessoa

Computer Architecture

Exercise sheet nº6

Goals:

- Processor optimization using MIPS pipelining

1. Consider the following times (in picoseconds) spent in the several MIPS pipeline stages.

Instruction Class	Instruction Fetch	Register Read	ALU Operation	Data Access	Register Write	Total time	Occurrences
lw	200	100	200	200	100	800	25%
sw	200	100	200	200		700	10%
add, sub, and, or, slt	200	100	200		100	600	50%
beq	200	100	200			500	15%

   a. Identify the average time per instruction without pipeline and using pipeline.
   b. If the ALU stage time was reduced in 25%, will this affect the pipeline? If yes how much? If not why?
   c. If, instead, the ALU stage time was increased in 25%, will this affect the pipeline? Identify the average time per statement with and without pipeline. The final performance was affected?

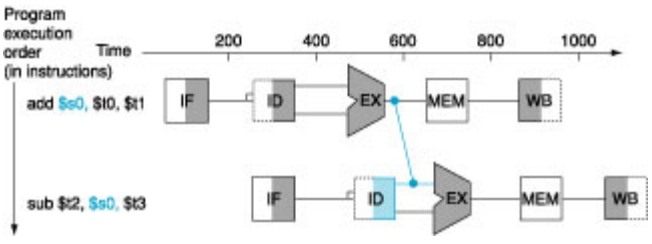
2. One CPU engineer needs to design a pipeline for a new CPU. A typical program to be executed by this CPU will have 10^6 instructions. Each instruction needs 100ps to be executed.

   a. How much time is needed to execute the typical program in a non-pipeline version of this CPU?

   b. Assume that current CPU pipelines have around 20 stages. Assuming that the designed pipeline was a perfect pipeline, and that the code will be executed thousands of times, what is the speedup compared with the previous non-pipeline version?

   c. Real pipelines are not perfect, since their implementation introduces some delay between stages. Will this delay affect the instruction latencies, the instructions rate per unit of time (throughput) or both? Why?

3. Using a diagram similar to the shown bellow, draw the needed lines to do forwarding in the 4 instructions bellow:




```
add $3, $4, $6
sub $5, $3, $2
lw $7, 100($5)
add $8, $7, $2
```

4. Identify all the dependencies in the following code. Which ones are data hazards that can be solved with forwarding? Which data hazards will cause the pipeline to stall?

```
add $3, $4, $2
sub $5, $3, $1
lw $6, 200($3)
add $7, $3, $6
```

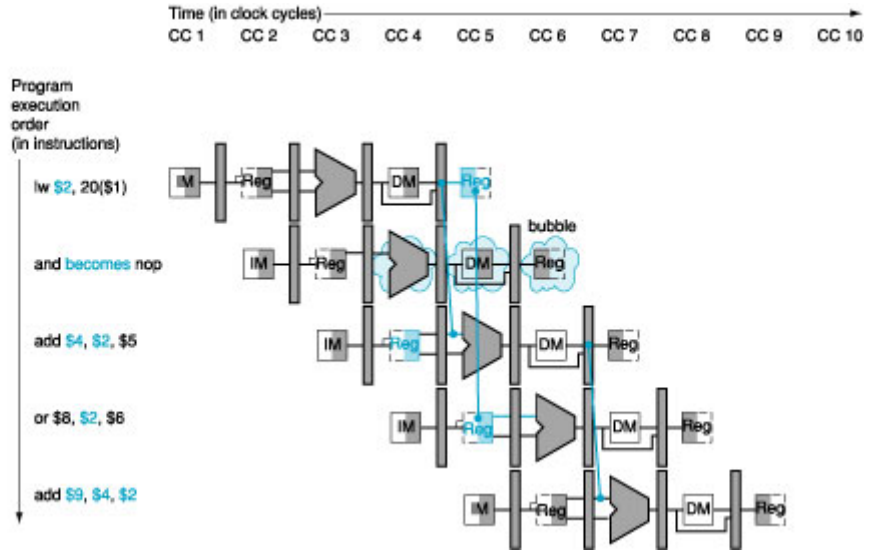
5. Rewrite the following code in order to *minimize* the performance (reorder the instructions to maximize the clock cycles needed to execute the code snippet and simultaneously to preserve the obtained result of the original version):

```
Loop: lw $2, 100($6)
      lw $3, 200($7)
      add $4, $2, $3
      add $6, $3, $5
      sub $8, $4, $6
      lw $7, 300($8)
      beq $7, $8, Loop
```

6. Consider the following code executed in a pipelined architecture:

```
lw $4, 100($2)
sub $6, $4, $3
add $2, $3, $5
```

   a. How many clock cycles are needed to execute this code?
   b. Show all the dependencies to be solved and draw a similar diagram to the one presented bellow to illustrate how the code will be executed (incorporate pipeline stalls and forwarding)



7. How can you modify the following code in order to use the delayed branch slot?

```
Loop:   lw $2, 100($3)
        addi $3, $3, 4
        beq $3, $4, Loop
```

8. Consider the following C code:

```
int array[10]={1,3,5,7,9,11,13,15,17,19}

main()
{
      int i=0,s=0;
      do{
            s+=array[i];
            i++;
      }while(i<10)

      printf("valor=%d\n",s);
}
```

a. Convert it to MIPS assembly. Execute it in the MIPS simulator and observe the output value.
b. Suppose that the code will be executed in a pipelined CPU. Rewrite it (introducing **`nop`** in relevant places) to be correctly executed with "branch delay slot" and "load delay slot".
c. Change the MIPS simulator settings to use "branch delay slot" and "load delay slot", execute the changed code and observe the output value.
d. Code from question b) can be optimized to use "branch delay slot" and "load delay slot" in a useful way? If yes, execute the necessary changes (justifying) and verify the program correction executing it again. If not justify it conveniently.

Bibliography:

[1] Patterson & Hennessy – *Computer Organization and Design: The hardware/software interface 4rd Ed* – MKP 2009.