Esta ficha foi elaborada para acompanhar o estudo da disciplina de Arquitetura de Computadores Avançada (a4s1) ou para complementar a preparação para os momentos de avaliação finais da mesma. Num segundo ficheiro poderás encontrar um conjunto de propostas de solução aos exercícios que estão nesta ficha. É conveniente relembrar que algum conteúdo destes documentos pode conter erros, aos quais se pede que sejam notificados pelas vias indicadas na página web, e que serão prontamente corrigidos, com indicações de novas versões.

1. Considera a seguinte sequência de código abaixo (Assembly para MIPS64):

```
loop:   l.d     f2, 0(rx)
i0:     div.d   f8, f2, f0
i1:     mult.d  f2, f6, f2
i2:     l.d     f4, 0(ry)
i3:     add.d   f4, f0, f4
i4:     add.d   f10, f8, f2
i5:     addi    rx, rx, 8
i6:     addi    ry, ry, 8
i7:     s.d     f4, 0(ry)
i8:     sub     r20, r4, rx
i9:     bnz     r20, loop
```

Consideremos também, e para o efeito, que as instruções de load têm uma latência de 4 ciclos, as instruções de store, branches e somas em vírgula flutuante têm uma latência de 1 ciclo, a instrução de multiplicação em vírgula flutuante têm uma latência de 5 ciclos, as instruções de divisão em vírgula flutuante têm uma latência de 12 ciclos e as operações aritméticas e lógicas inteiras têm uma latência de 0 ciclos. Neste cenário:

a) Qual é o desempenho base, em número de ciclos (por iteração do loop), da sequência de código acima se nenhuma instrução puder ser iniciada sem que uma anterior seja completada? Ignorando os tempos de fetch e de descodificação (decode), consideremos ainda que a execução das instruções não para à conta de uma instrução seguinte, mas que as instruções são lançadas uma por ciclo e que o branch é taken, isto é, existe um delay slot de um ciclo de branch.

Cada instrução, por si, tem um tempo de um ciclo para o seu lançamento. Assim, temos um número de ciclos inicial, de 11. A estes 11, por cada instrução, devemos adicionar a sua latência natural, conforme o código em baixo:

```
                                no. de ciclos
loop:   l.d     f2, 0(rx)           1 + 4
i0:     div.d   f8, f2, f0          1 + 12
i1:     mult.d  f2, f6, f2          1 + 5
i2:     l.d     f4, 0(ry)           1 + 4
i3:     add.d   f4, f0, f4          1 + 1
i4:     add.d   f10, f8, f2         1 + 1
i5:     addi    rx, rx, 8           1 + 0
i6:     addi    ry, ry, 8           1 + 0
i7:     s.d     f4, 0(ry)           1 + 1
i8:     sub     r20, r4, rx         1 + 0
i9:     bnz     r20, loop           1 + 1
                       total:     11 + 29 = 40
```

Com as latências calculadas, temos então um total de número de ciclos de 40.

b) Quantos ciclos de execução o corpo do "loop" se a sequência de código necessitaria se o pipeline detetasse dependências de dados verdadeiras e só parasse a execução (stall) nesses instantes, ao invés de parar tudo cegamente só porque uma determinada unidade funcional se encontra ocupada? Mostra o código com a indicação de stalls onde necessário de forma a acomodar as tais latências.

Ora, uma instrução com latência de 2 ciclos necessita de dois ciclos em stall ou, por outro lado, uma instrução de um ciclo (sem latência), necessita de 0 ciclos de stall. Assim sendo, verifiquemos o código abaixo:

```
                                no. de ciclos
loop:   l.d     f2, 0(rx)           1 + 4
        <stall>
        <stall>
        <stall>
        <stall>
i0:     div.d   f8, f2, f0          1 + 12
i1:     mult.d  f2, f6, f2          1 + 5
i2:     l.d     f4, 0(ry)           1 + 4
        <stall devido ao l.d>
        <stall devido ao l.d>
        <stall devido ao l.d>
        <stall devido ao l.d>
i3:     add.d   f4, f0, f4          1 + 1
```

```
        <stall devido ao add.d>
        <stall devido ao div.d>
        <stall devido ao div.d>
        <stall devido ao div.d>
        <stall devido ao div.d>
i4:     add.d   f10, f8, f2         1 + 1
i5:     addi    rx, rx, 8           1 + 0
i6:     addi    ry, ry, 8           1 + 0
i7:     s.d     f4, 0(ry)           1 + 1
i8:     sub     r20, r4, rx         1 + 0
i9:     bnz     r20, loop           1 + 1
        <stall devido ao branch delay slot>
```

Como agora a sequência de instruções poderá ser executada sem ter noção de deteção de hazards, basta contar o número de linhas (instruções - sendo que as de "stall" são "nop") e obtemos um valor de 25 iterações.

2. O seguinte código é denominado de DAXPY e a sua operação fundamental é a redução de uma matriz por eliminação com o método de Gauss, neste caso, para um vetor de tamanho 100. Considere-se que r1 contém o endereço-base do array X e que r2 contém o endereço-base do array Y:

```
        daddiu  r4, r1, 800     ; endereço-base de X
for:    l.d     f2, 0(r1)       ; (f2) = X[i]
        mul.d   f4, f2, f0      ; (f4) = a*X[i]
        l.d     f6, 0(r2)       ; (f6) = Y[i]
        add.d   f6, f4, f6      ; (f6) = a*X[i] + Y[i]
        s.d     f6, 0(r2)       ; Y[i] = a*X[i] + Y[i]
        daddiu  r1, r1, 8       ; incremento de índice de X
        daddiu  r2, r2, 8       ; incremento de índice de Y
        dsltu   r3, r1, r4      ; teste: prosseguir no ciclo?
        bnez    r3, for         ; ciclo em for se necessário
```

Assumam-se, também, que temos as seguintes latências, por par (unidade funcional, instrução): (multiplicador FP, operação ALU FP) = 6 ciclos de relógio; (somador FP, operação ALU FP) = 4 ciclos de relógio; (multiplicador FP, operação de store) = 5 ciclos de relógio; (somador FP, operação de store) = 4 ciclos de relógio; (unidade inteira, operações inteiras e loads) = 2 ciclos de relógio.

a) Considera um pipeline comum de cinco andares e mostra a que é que o "loop" se assemelharia tanto antes como depois do agendamento para ambas operações de vírgula flutuante e branch delays, incluindo quaisquer stalls ou ciclos de inatividade (idle cycles). Qual é o tempo de execução (em ciclos) por elemento do vetor de resultado Y, tanto na sua versão não-agendada, como na agendada? Quão mais rápido deverá ser o relógio para que o processador, por si, consiga atingir o desempenho da versão agendada pelo compilador? (Ignora quaisquer efeitos de aumento do ciclo de relógio no desempenho da memória do sistema)

Numa versão não-agendada, apenas temos de efetuar o desmembramento do ciclo do programa DAXPY. Assim sendo, e considerando as latências, temos o seguinte código:

```
ciclo:
1               daddiu  r4, r1, 800     ; endereço-base de X
2       for:    l.d     f2, 0(r1)       ; (f2) = X[i]
3               stall
4               mul.d   f4, f2, f0      ; (f4) = a*X[i]
5               l.d     f6, 0(r2)       ; (f6) = Y[i]
6               stall
                stall                   ; idle
                stall                   ; idle
                stall                   ; idle
7               add.d   f6, f4, f6      ; (f6) = a*X[i] + Y[i]
8               stall
9               stall
10              stall
11              s.d     f6, 0(r2)       ; Y[i] = a*X[i] + Y[i]
12              daddiu  r1, r1, 8       ; incremento de índice de X
13              daddiu  r2, r2, 8       ; incremento de índice de Y
14              dsltu   r3, r1, r4      ; teste: prosseguir no ciclo?
15              stall
16              bnez    r3, for         ; ciclo em for se necessário
17              stall
```

Como podemos ver, contando o número de instruções do código, o tempo de execução (em ciclos), da versão não-agendada é de 16 ciclos. Com uma versão agendada, significa que há uma reordenação das instruções, sendo que instruções sem dependências entre si poderão ser movidas para zonas de código em que haja um melhor aproveitamento do seu tempo de execução. Assim sendo, conseguimos chegar ao código seguinte, onde levantamos todas as instruções mais morosas para o início de execução, aproveitando as suas não-dependências de dados.

```
ciclo:
1               daddiu  r4, r1, 800     ; endereço-base de X
2       for:    l.d     f2, 0(r1)       ; (f2) = X[i]
3               l.d     f6, 0(r2)       ; (f6) = Y[i]
4               mul.d   f4, f2, f0      ; (f4) = a*X[i]
5               daddiu  r1, r1, 8       ; incremento de índice de X
6               daddiu  r2, r2, 8       ; incremento de índice de Y
7               dsltu   r3, r1, r4      ; teste: prosseguir no ciclo?
                stall
                stall
8               add.d   f6, f4, f6      ; (f6) = a*X[i] + Y[i]
                stall
                stall
9               bnez    r3, for         ; ciclo em for se necessário
10              s.d     f6, 0(r2)       ; Y[i] = a*X[i] + Y[i]
```

Olhando para o código acima podemos ver que o tempo de execução é de 10 ciclos de relógio, sendo que podemos concluir que esta versão é 60% mais rápida que a original, pelo que esta é a percentagem de melhoria que o hardware terá de suportar para obter um mesmo desempenho.

3. Descreve uma forma de exploração de cada um dos dois tipos de paralelismo juntamente com um desafio em fazê-la.

a) Paralelismo de Instruções;

Uma forma de explorar o paralelismo de instrução é estudando a possibilidade de processamento fora-de-ordem, isto é, tentar encontrar as limitações da execução de instruções por uma ordem diferente daquela com que as instruções estão presentes numa dada sequência na memória (verificação de dependências de dados verdadeiras), tendo, possivelmente, registos suficientes para possíveis renomeações.

b) Paralelismo de Dados;

O paralelismo de dados poderá ser explorado usando arquiteturas vetorizadas ou SIMD. Uma limitação (e consequente desafio) é a dificuldade de otimização de código que já era executado em perfeitas condições de uma forma sequencial, agora, num conjunto de processos em paralelo.

4. Consideremos uma máquina com um pipeline clássico de cinco andares MIPS que usa predição de branch sem delay slots e que possui uma taxa de mispredict penalty de 3 ciclos de execução. Considerando também que em uma de cada cinco instruções encontra-se uma instrução de branch para um dado programa, para os quais 80% são bem previstos pelo nosso preditor:

a) Quantos ciclos demora a executar N instruções?

Ora, o número de ciclos de relógio por instrução (CPI) é então 1 + M, sendo M o prejuízo de má predição (wrong prediction penalty). Sendo que a taxa de mispredict é de 3 e a percentagem de falhas em predições no nosso programa é de 1 - 80% = 20%, para 1 em cada 5 instruções, temos que M = (1/5 x 0.2 x 3) = (0.2 x 0.2 x 3) = 0.12. Logo, o CPI é igual a 1 + 0.12 = 1.12. Sendo este o CPI, o número de ciclos que demora a executar N instruções é simplesmente 1.12 x N.

b) Assumindo que temos um processador Intel Pentium 4 (que possui um pipeline de 20 andares), agora temos uma inacreditável taxa de mispredict penalty de 19 ciclos. Qual deverá ser o rácio de predição de branch de forma a que tenhamos o mesmo desempenho que no pipeline de cinco andares clássico do MIPS, tal como vimos em a)?

De forma semelhante à alínea a), tendo que o CPI é de 1.12, este seria formado por 1 instrução mais os 19 ciclos de má predição (mispredict penalty) numa percentagem (1-x) de falhas em predições no nosso programa, para 1 em cada 5 instruções.

$1.12 = 1 + (1/5 \times (1-x) \times 19) \Leftrightarrow 0.12 = 3.8 \times (1-x) \Leftrightarrow 1 - x = 0.031 \Leftrightarrow x = 0.9684$

A taxa de predição correta para o Pentium 4 é de cerca de 96.84%, de forma a obter um desempenho semelhante ao do MIPS.

5. Consideremos um pipeline MIPS clássico de cinco andares e o código abaixo a ser executado neste:

```
linha
1.      lw      r5, 20(r0)
2.      add     r4, r0, r0
3.      sw      r5, 100(r4)
4.      add     r5, r5, r5
5.      sw      r5, 104(r4)
```

```
6.      add     r5, r5, r5
7.      sw      r5, 106(r4)
8.      add     r5, r5, r5
9.      sw      r5, 108(r4)
```

a) Identifica potenciais hazards no código acima. Para cada tipo de hazard (estrutural, de dados e de controlo), apresenta uma ocorrência. Detalha a tua resposta com a linha(s) envolvidas no hazard, e sugere uma correção. Mesmo que todas as linhas tenham um hazard, identifica apenas um de cada tipo, assumindo que tais hazards estão presentes no código.

Começando pelos hazards de controlo, para os encontrar, devemos procurar branches. Como não há instruções de branch no código, então não há hazards de controlo.

Como hazard de dados, na linha 3 podemos ver um hazard do tipo RAW (Read After Write). O que acontece nesta linha é que estamos a tentar guardar em memória o conteúdo de um registo que ainda não possui os seus dados corretos. Uma possível correção seria efetuar o forwarding da fase de acesso à memória (MEM) da instrução da linha 1 para a fase de execução (EX) da instrução corrente (linha 3).

Como hazard estrutural (na linha 4), olhando para a linha 7 podemos verificar uma situação em que tentamos ler e escrever ao mesmo tempo no mesmo registo, dado que quando esta tenta ler, na linha 4 está a tentar escrever-se um valor. Uma possível correção é através da unidade de interlocking (inserção de stalls).

b) Calcula o número de ciclos por instrução (CPI) do código acima, assumindo que não há unidade de forwarding (considera também que a última instrução é sw r5, 112(r4)).

O número de ciclos de relógio por instrução (CPI) é igual ao quociente entre o número de ciclos e o número de instruções. Já sabemos que o número de instruções é 9, mas qual é o número de ciclos. Ora, analisemos a execução abaixo, onde as fases do pipeline estão identificadas pelos códigos F, D, E, M, W e os stalls estão identificados com s.

```
linha  1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20  21
1.     F   D   E   M   W
2.         F   D   E   M   W
3.             F   D   s   s   E   M   W
4.                 F   s   s   D   E   M   W
5.                         F   D   s   s   E   M   W
6.                             F   s   s   D   E   M   W
7.                                 F   D   s   s   E   M   W
8.                                     F   s   s   D   E   M   W
9.                                             F   D   E   s   s   M   W
```

Pela execução acima podemos verificar que o número de ciclos é 21. Assim sendo, temos que o CPI é igual a 21/9 ≈ 2.33.

c) Corrige as dependências de dados com forwarding e identifica, caso existam dependências não passíveis de serem resolvidas com técnicas de forwarding ou interlocking. Considera, novamente, que a última instrução é sw r5, 112(r4). Calcula novamente o CPI.

Na execução em baixo, à direita estão os forwardings identificados com os pares "forwarding de (linha, fase) para (linha, fase)".

```
linha  1   2   3   4   5   6   7   8   9   10  11  12  13
1.     F   D   E   M   W
2.         F   D   E   M   W
3.             F   D   E   M   W                       forwarding de (2,E) para (3,E)
4.                 F   D   E   M   W                   forwarding de (1,M) para (4,E)
5.                     F   D   E   M   W               forwarding de (4,E) para (5,M)
6.                         F   D   E   M   W           forwarding de (4,E) para (5,E)
7.                             F   D   E   M   W       forwarding de (6,E) para (7,M)
8.                                 F   D   E   M   W   forwarding de (6,E) para (8,E)
9.                                     F   D   E   M   W   forwarding de (8,E) para (9,M)
```

Novamente, o cálculo do CPI é agora 13/9 ≈ 1.44. Não há mais hazards de dados para corrigir (a técnica de forwarding é suficiente).

6. Poderá acontecer algum hazard do pipeline envolvendo o registo R0 (zero)?

Sim. Como o registo R0 não é um registo de escrita, então os compiladores deverão ser espertos o suficiente para nunca usar tal registo como destino de uma instrução, de forma a não provocar quaisquer erros que possam envolver o registo R0. O hazard RAW não é possível então, porque não existe tal coisa como escrita em R0, pela mesma razão que não existem hazards WAR e WAW. Em termos de hazards de controlo, poderão existir tais hazards caso num ciclo de 'branch not equal to zero', R0 é executado e nós estivermos a assumir que é taken, ou quando estamos perante um ciclo de 'branch equal to zero' e esperamos por um resultado not taken.