Nota: uma resposta errada nas questões 1 a 6 desconta 0.33 valores

1. [1,0 valores] - Considere código C + OpenMP apresentado abaixo:

```
#pragma omp parallel
{  int i, first=false, tid = omp_get_thread_num ();
   double T;
   printf ("Thread %d starting\n", tid);
   T = omp_get_time ();
   #pragma omp for
     for (i=0; i < 300000 ; i++) do_work(i);
   #pragma omp single
   {
     first = true;
     printf ("Thread %d work done\n", tid);
   }
   if (first) printf ("1st finished in %.0lf us\n", (omp_get_wtime()-T)*1e6);
   printf ("Thread %d finishing\n", tid);
}
```

Para uma execução com 3 *threads* indique qual dos *outputs* abaixo é possível.

☐	Thread 1 starting Thread 0 starting Thread 0 work done 1st finished in 7 us Thread 2 starting Thread 2 finishing Thread 1 finishing Thread 0 finishing		☐	Thread 1 starting Thread 0 starting Thread 2 starting Thread 0 work done Thread 2 finishing 1st finished in 7 us Thread 1 finishing Thread 0 finishing
☐	Thread 1 starting Thread 0 starting Thread 2 starting Thread 2 finishing Thread 0 work done 1st finished in 7 us Thread 1 finishing Thread 0 finishing		☐	Thread 1 starting Thread 0 starting Thread 2 starting Thread 0 work done Thread 0 finishing 1st finished in 7 us Thread 1 finishing Thread 2 finishing

2. [1,0 valores] - Complete a afirmação abaixo :

"O ganho de desempenho obtido com a vectorização de código, relativamente à respectiva versão escalar, deve-se

☐ à diminuição do número médio de ciclos por instrução (CPI)."

☐ à diminuição do número total de operações matemáticas executadas sobre os dados."

☐ a acessos mais rápidos à memória, devidos à maior localidade espacial imposta pelas intruções de `mov` vectoriais."

☐ à diminuição do número total de instruções executadas (#I)."

Nome: _____ Número:_____

3. [1,0 valores] - O *loop unrolling* tem potencial para disponibilizar mais instruções para execução em paralelo num contexto de superescalaridade. Para o código abaixo seleccione a opção de *unrolling* que disponibiliza potencialmente mais *instruction level parallelism*.

```
int a[SIZE], i, sum=0;
for (i=0; i < SIZE ; i++) sum +=a[i];
```

☐	`int a[SIZE], i, sum=0;`
	`for (i=0; i < SIZE ; i+=2)`
	`{ sum +=a[i];`
	`  sum +=a[i+1]; }`
			☐	`int a[SIZE], i, sum=0, sum_a=0;`
				`for (i=0; i < SIZE ; i+=2)`
				`{ sum_a +=a[i];`
				`  sum +=a[i+1]; }`
				`sum += sum_a;`

☐	`int a[SIZE], i, sum=0;`
	`for (i=0; i < SIZE ; i+=4)`
	`{ sum +=a[i];`
	`  sum +=a[i+1];`
	`  sum +=a[i+2];`
	`  sum +=a[i+3]; }`
			☐	`int a[SIZE], i, sum=0;`
				`for (i=0; i < SIZE ; i+=4)`
				`{ sum += a[i] + a[i+1];`
				`  sum += a[i+2] + a[i+3]; }`

4. [1,0 valores] - O código "`for (i=1 ; i<S ; i+=1) a[i]=a[i-1] / (i>10 ? 2.:3.);`" não vectoriza devido…

   ☐ a uma dependência de dados RAW entre iterações.

   ☐ aos dados processados não se encontrarem em posições consecutivas de memória.

   ☐ à possibilidade de *aliasing* entre `a[i]` e `a[i-1]`.

   ☐ à estrutura condicional incluída dentro do ciclo `for`.

5. [1,0 valores] – Um programa P, escrito em OpenMP, executado com uma única *thread* numa máquina com uma frequência de 2 GHz, apresenta um CPI de 1.5 e um tempo de execução de 1.5 segundos.
   O mesmo programa executado em 10 núcleos (com 10 *threads*) executa 10% mais instruções e exibe um *speed up* de 7.5. O $CPI_{percepcionado}$ é

☐	0.75	☐	0.182
☐	0.212	☐	0.15

6. [1,0 valores] – A eficiência exibida na alínea anterior é:

☐	7.5	☐	10%	☐	75%	☐	50%

7. [2,0 valores] - Considere um processador superescalar com 2 unidades funcionais (UF):

UF1 (Op) – realiza operações lógicas e aritméticas sobre inteiros;

UF2 (LS + B) – realiza acessos à memória (*Load/Store*) e saltos (*branches*).

Considere que cada uma destas unidades funcionais executa **uma instrução por ciclo do relógio** (isto é, não há nenhuma operação que exija mais do que um ciclo do relógio na respectiva UF). Considere ainda o seguinte excerto de código:

```
I1:   movl (%ebx, %edx, 4), %esi
I2:   addl %esi, %eax
I3:   incl %edx
I4:   decl %ecx
I5:   jnz I1
```

Preencha, para a primeira iteração do ciclo, a tabela 1 considerando um escalonamento *static in-order scheduling*:

- Para preencher a tabelas indique, para cada instrução, qual a unidade funcional em que é escalonada e em que ciclo do relógio. Calcule também o CPI exibido por essa primeira iteração.
- sugere-se que numa folha à parte preencha primeiro uma tabela com o layout apresentado a seguir e apenas depois converta para o formato pedido (note que nesta tabela nas células indica qual a instrução correspondente):

Ciclo	U1	U2
1		
2		
⋮		

	Tabela 1 - *static in-order*	
	UF (1 ou 2)	**Ciclo**
I1		
I2		
I3		
I4		
I5		
	CPI =	

Use o espaço abaixo para alguma justificação que lhe pareça pertinente.

8. [2,0 valores] - O código apresentado abaixo, que explora *Thread Level Parallelism* recorrendo ao OpenMP, pretende calcular a soma de alguns elementos de cada linha `i` de uma matriz (elementos das colunas `1` a `i`) e armazenar o resultado no primeiro elemento dessa linha (`a[i][0]`) :

```
#define W 400000
int a[W][W];
int sum, i, j;

...
```

```
#pragma omp parallel for
for (i=0 ; i < W ; i++) {
    sum = 0;
    for (j=1 ; j <= i ; j++) sum += a[i][j];
    a[i][0] = sum;

}
```

O resultado da execução deste programa com múltiplas *threads* é indeterminado, pois contém alguns erros semânticos. Identifique esses erros e diga como os corrigiria.

NOTA: Os erros semânticos não estão relacionados com o desempenho, mas sim com a correcção do programa.