1 - B

```
int main(){
  char a[]="135";
  char *p1, **p2;
  //variável a contém endereço do primeiro elemento do array
  ///(caracter '1')
  p1 = a+1; //p1 = a + 1 = &a[1]
  p2 = &p1;//p2 = &p1 implica *p2 = p1 = &a[1]
  //implica **p2 = *p1 = *(&a[1]) = a[1]

  printf("%c, %c, %c \n",a[1], *p1,*(*(p2)+1));
  //*p1 = *(&a[1]) = a[1] = 3
  //*(*(p2)+1) = *((*p2)+1) = *(p1+1) = *(&a[1]+1) =
  //= *(a + 1 + 1) = *(a+2) = *(&a[2]) = a[2] = 5
  return 0;
}
```

2 - C

CPI penaltys = CPI real - CPI ideal = 4 - 2 = 2

CPIpenatlys=%missInstrucoes*penaltyIntrucoes+%acessosDados*%missDados*penaltyDados

2 = (1-0.85)*8 + %acessoDados*0.2*10

2 = 1.2+2*%acessoDados

0.8/2 = %acessoDados

%acessoDados = 0.4 = 40%

3 -

0x80402010 = 0b0000000.01000000.00100000.00010000
0x00FFFF00 = 0b0000000.11111111.11111111.00000000
AND e OR em assembly são aplicados bit a bit, então:
0x80402010 AND 0x00FFFF00 =
= 0b0000000.01000000.00100000.00010000 AND
  0b0000000.11111111.11111111.00000000
= 0b00000000.01000000.00100000.00000000 = 0x00402000

0x00402000 OR 0x0040 =
= 0b00000000.01000000.00100000.00000000 OR
  0b00000000.00000000.00000000.01000000
= 0b00000000.01000000.00100000.00000100 = 0x00402040

SRL (Shift right logical) de 0x00402040 por 8 casas faz-se:
= 0x00402040 >> 8 =
= 0b00000000.01000000.00100000.000 00100 >> 8 =
= 0b00000000.01000000.00100000 = 0x004020 = 0x00004020

Tendo uma visão mais macroscópica das operações:

- 0x80402010 AND 0x00FFFF00 isola os 2 bytes centrais (res: 0x00402000)
- 0x00402000 OR 0x00000040 complementa os bits a 0 com os do segundo operando (res: 0x00402040)
- 0x00402040 >> 8 desloca todos bytes 8 bits para a direita, como cada caractere hexadecimal corresponde a 4 bits, na prática removemos os 2 caracteres mais à direita (res: 0x00004020)

Extra: a utilidade destas operações é exemplificada no trabalho prático 2, alínea B

4 - D

Todos os argumentos de uma chamada de função são escritos na stack, como tal não existe um limite de argumentos definido pelo hardware. Os primeiros 4 argumentos de uma função são por defeito inscritos nos registos a0 a a3 o que permite aumentar a performance para funções com poucos argumentos e/ou que acedem frequentemente aos primeiros, uma vez que o acesso a registos é mais rápido que o acesso a memória.

O ponteiro $sp é movido cada vez que uma função é chamada ou retorna.

Enquanto que o descrito na alínea C é possível, não respeita a convenção do MIPS. Deve-se utilizar para o efeito os registos s.

O registo $ra mantém a informação da linha de código que deve correr após o retorno de uma função, sempre que uma função chama outra deve guardar o seu registo $ra para que possa retornar mais tarde. Para exemplificar o funcionamento disto acompanhe o funcionamento do código abaixo. Tenha em conta que o $ra regista o endereço da instrução a ser corrida e não da linha propriamente dita.

```
void funcaoB(){
  //funcao chamada por funcaoA na linha 18, $ra = 19
   //vou chamar funcao entao ponho na stack o valor 19 e $ra passa a ser linha seguinte, $ra = 8
  printf("hmhm\n");
  //funcao voltou, ra volta a ser $ra = 19 (retirado da stack)
  return;
}

void funcaoA(){
  //funcao chamada por main na linha 26, $ra = 27
   //vou chamar funcao entao ponho na stack o valor 27 e $ra passa a ser linha seguinte, $ra = 16
  printf("blablabla\n");
  //funcao voltou, ra volta a ser $ra = 27 (retirado da stack)
    //chamar funcao entao ponho na stack o valor 27 e $ra passa a ser linha seguinte, $ra = 19
  funcaoB();
  //funcao voltou, ra volta a ser $ra = 27 (retirado da stack)
  return;
}

int main(){
  //funcao inicial, $ra está vazio
  //vou chamar funcao $ra passa a ser linha seguinte, $ra = 27
  funcaoA();
  //funcao voltou, ra volta a ser vazio
  return 0;
}
```

5 - B

Carregamos em $t2 o valor 100 a partir da variável my_ptr.
A instrução “slti” coloca $t3 a 0 porque $t2 < 10 é falso.
O valor de $t3 é colocado na variável my_ptr.

6 - A

Sempre que uma função é chamada cria uma entrada na stack onde guarda todos os dados relevantes para a mesma bem como variáveis locais declaradas durante a execução da função. Assim que retorna esta entrada na stack é descartada, podendo ser sobrescrita.

7 - C

Variáveis declaradas fora de qualquer função são globais, todas as variáveis globais são armazenadas na zona de dados estáticos.

Variáveis declaradas dentro de funções são sempre colocadas na stack (pilha). Após o retorno da função todas as variáveis armazenadas na stack dessa função são eliminadas.

Memória alocada com calloc ou malloc é armazenada na heap. O endereço da memória alocada pode ser atribuído a variáveis do tipo ponteiro. Enquanto que a variável ponteiro é armazenada na stack, a memória apontada e atribuída dinamicamente está sempre localizado na heap e perdura após o retorno da função.

Atribuir a um ponteiro o endereço de uma qualquer variável não significa que a memória apontada esteja na heap uma vez que a variável não é alocada dinamicamente.

Ex:

```
Int i, *aux;
aux = &i;
```

As variáveis i e aux estão armazenadas na pilha sendo que aux aponta para uma zona de memória da pilha.

8 - D

9 - B

256kB = $2^8 * 2^{10} = 2^{18}\ bytes$

$2^{18}/512 = 2^{18}/2^9 = 2^{18-9} = 2^9$

Uma direct mapped cache de 256kB e blocos de 512 bytes seria indexada por 9 bits. Esta é indexada por 8 bits, trata-se portanto de uma N-way set-associative cache.

$9 - log_2 N = 8 \Rightarrow log_2 N = 1 \Rightarrow N = 2$

É portanto uma 2-way set-associative cache.

De notar que uma direct mapped cache não é mais que uma 1-way set-associative cache.

10 - A

Na prática todos os elementos do array são substituídos pelo seu quádruplo. A explicação do código encontra-se abaixo.

```
.data
tam: .word 10
tab: .word 1,2,3,4,5,6,7,8,9,10
#notar que este array é constituido por words, iterar implica andar de 4 em 4 bytes.
.text
la $t0,tab  #t0 aponta para o primeiro elemento do array
la $t1,tam
lw $t2,0($t1)   #t2 contém o valor da variável tam
add $t3,$0,$0 #t3 inicializado a 0
ciclo:
beq $t2,$t3,out  #caso t2 seja igual a t3 salta para out, terminando o ciclo
lw $t4,0($t0)  #para t4 é carregado o valor apontado por t0
sll  $t4,$t4,2  #t4 sofre um shift left de 2 bits. é o equivalente a multiplicar por 2^2
sw $t4,0($t0)  #o valor apontado por t0 é substituido por t4
addi  $t3,$t3,1  #t3 é incrementado
addi  $t0,$t0,4  #t0 passa a apontar para o proximo elemento do array
j ciclo
out:
```

11 - C

Ocorre nova iteração do ciclo se t0 != 0, sendo t0 = 1 se s3 < s4 ou 0 caso contrário, também ocorre nova iteração do ciclo se t1 = 0, sendo t1 = 1 se s4 < s5 ou 0 caso contrário.

A primeira condição traduz-se em simplesmente s3 < s4, a segunda condição traduz-se em s4 >= s5. Basta uma das condições ser verificada para ocorrer o ciclo portanto usamos o operador “ou” (||). As condições devem ser verificadas nesta ordem.

12 - D

Notar que tab é um array de int, ocupando portanto 4 bytes.

ptr = tab + 3 = &tab[3]

Sendo que a tabela tab está armazenada em 0x7F7FE580 e o array é de int, então:

ptr = tab + 3 = 0x7F7FE580 + 3*4 = 0x7F7FE580 + 0xC = 0x7F7FE58C.

*ptr = *(tab + 3) = *(&tab[3]) = tab[3] = 7

*ptr - 2 = *(tab + 3) - 2 = *(&tab[3]) - 2 = tab[3] - 2 = 7 - 2 = 5

*(ptr-2) = *(tab + 3 - 2) = *(&tab[3-2]) = tab[1] = 3

13 - B

Todas as instruções estão ativas nas fases 1 e 2.

Na fase 3 estão ativas instruções que requerem fazer qualquer cálculo aritmético ou lógico (somas, subtrações, multiplicações, e, ou, menor, maior, igual etc).

Na fase 4 estão ativas funções que acedem à memória, isto é a um endereço. Acesso a registo não é acesso à memória.

Na fase 5 estão funções que guardam algo num registo tal como o resultado de uma operação aritmética.

“slti” e “sll” estão ativas em todas as fases excepto a 4. “lbu” está ativo em todas as fases. “sb” só não está ativa na fase 5, uma vez que não altera o valor de qualquer registo, escrevendo apenas em memória.

14 - A

AAT = tempoCache + %miss*tempoMemoria
15 = 50/5 + %miss*50
15-10 = %miss*50
%miss = 5/50 = 0.1 = 10%
hitRate = 1 - %miss = 1 - 0.1 = 0.9 = 90%

15 - C