1 - A

Código para teste:

```
int num = 5;
int *ptr = &num;
printf("%d %x\n", *ptr, *ptr);
printf("%d %x\n", &ptr, &ptr);
printf("%d %x\n", &num, &num);
printf("%d %x\n", ptr, ptr);
```

output:

```
5 5
-13296 ffffcc10
-13284 ffffcc1c
-13284 ffffcc1c
```

Explicação:

Valor da variável ptr é o endereço de memória onde está guardada a variável num.

*ptr trata o valor de ptr como um endereço de memória, ao qual acede e recupera o valor guardado nesse endereço.

Como ptr = endereço de num, *ptr é o valor da variável num.

2 - D

O array é declarado com .byte portanto cada elemento é constituído por apenas 1 byte e aceder ao elemento seguinte implica deslocar-se 1 byte.

Aceder ao elemento 70, que é o 7º elemento implica portanto deslocar-se 6 bytes.

De forma a escrever na memória utiliza-se a função store, como cada valor é representado por 1 byte usa-se sb (Store byte).

3 - B

10% das instruções não estão em cache e portanto 10% das instruções demoram pelo menos 15 ciclos extra.

25% das instruções acedem a dados, 20% destes acessos são a dados que não estão em cache, portanto 0.25*0.2 = 5% das instruções têm um penalty adicional de 20 ciclos

Portanto o processador sofre penaltys de CPI iguais a 0.1*15 + 0.05 *20 = 1.5+1 = 2.5

Caso todos os acessos à memória envolvessem apenas a cache este penalty não existiria. O CPI real é de 5, sem este penalty seria de 5-2.5 = 2.5 CPI

4 - C

```
.data
frase: .asciiz "Bom dia!?" #cada caracter representdo poa 1 byte
.text
la $t0,frase #frase comtém endereço de primeiro caracter (B)
lbu $t1,2($t0)    #é carregado para t1 o valor 2 bytes à frente  de B (m)
lbu $t2,5($t0)    #é carregado para t1 o valor 5 bytes à frente  de B (i)
slt $t3,$t1,$t2    #t1 < t2 (m < i) = false portanto $t3 fica a 0
bne $t3,$0,salta    #$t3 == 0 portanto não ocorre o salto
lbu $t1,0($t0) #é carregado para t1 o valor 0 bytes à frente  de B
#(o proprio B)
salta:    #linha ignorada
addi $t1,$t1,1 #B = B + 1 -> B = C
```

5 - B

Speedup = 1/(missrate + hitrate/Ge) 4 = 1/(0.2+0.8/Ge) 0.25 = 0.2 + 0.8/Ge 0.05 = 0.8/Ge Ge = 0.8/0.05 = 16	Ge = memoryTime/cacheTime 16 = 32/cacheTime cacheTime = 2	AAT = 32*(1-hitrate) + (cacheTime) AAT = 32 * 0.2 + 2 = 6.4 + 2 = 8.4

6 - D

```
.data
tab: .word 1,2,3,4,5,6,7,8,9,0  #array definido com inteiros do tipo word (4 bytes)
.text
la $t0,tab  #carrega em t0 endereço do primeiro elemento do array
li $t1,10  #t1 = 10
li $t7,2  #t7 = 2
ciclo:
beqz $t1,out  # salta quando t1 = 0
lw $t2,0($t0)  #carrega em t2 o valor no array apontado por t0
mul $t2,$t2,$t7  # t2 = t2 *t7 = t2*2
sw $t2,0($t0)  #guarda t2 na posicao apontada por t0
addiu $t0,$t0,8  #t0 aponta para o elemento do array 2 casas à frente
#(array de word 4 bytes, 2*4 = 8)
subi $t1,$t1,2  #t1 = t1 - 2
j ciclo #salta para label ciclo
out:
#vai comecar com t1 = 10 e em cada iteração remove 2, parando quando t1 = 0
#multiplica posições do array por 2, começando na posicao 0 e ignorando posicoes impares
#resultado final de tab é
#2 2 6 4 10 6 14 8 18 0
```

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 - A

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.

Para “srl $t1, $t0, 4” temos ativação nas fases 1 e 2, na fase 3 (operação t0 << 4) e na fase 5 escrevendo o resultado da operação referida no registo t1.

9 - B

Notar que as tabelas são de int, que corresponde a valores de 4 bytes (word), portanto iterar em assembly implica andar de 4 em 4 bytes.

Se $a1 contém endereço de A e $a2 contém endereço de B então na primeira iteração do assembly temos que 0($a1) corresponde a A[0] e 4($a2) corresponde a B[1].

Para guardar na posição A[1] devemos portanto aceder a 4($a1) e devemos passar ao próximo elemento fazendo $a1 = $a1 + 4.

A mesma lógica aplica-se às iterações seguintes.

10 - C

```
int main() {
  int tab[5] = {1, 2, 3, 4, 5};
  int *ptr = tab + 2; //aritmetica de ponteiros em C
  //ptr = tab + 2 é equivalente a ptr = &tab[2]
  //portanto ptr aponta para o elemento indice 2 do array
  printf("%d ", tab[3]); //imprime o
  printf("%d ", *ptr); //imprime o 3
  printf("%d ", *(ptr + 2)); //*(ptr +2)= *(&tab[2] +2)
  // = *(tab+2+2) = *(tab+4) = *(&tab[4]) = tab[4]
  //imprime 5
  printf("%x ", ptr + 1); //ptr + 1 = &tab[2] + 1 = tab + 3 = &tab[3]
  //imprime endereço de tab[3] em formato hexadecimal
  printf("%x \n", tab); //imprime endereço de tab em formato hexadecimal
  return 0;
}
```

Assumindo que o primeiro elemento de tab está armazenado no endereço de memória 0x00007000 então &tab[3] = 0x00007000 + 3*4 = 0x00007000 + C = 0x0000700C.

11 - C

t0 = s0 + s1	t0 = a + b
t1 = 2	t1 = 2
t2 = s2*t1	t2 = c*t1
t3 = t2-s3	t3 = t2-d
s4 = t0-t3	f = t0-t3

f = t0 - t3 -> f = (a+b) - (t2-d) -> f = (a+b) - ((c*t1)-d) -> f = (a+b) - ((c*2)-d)

12 - B

O gdb permite tudo o referido excepto gerar um novo executável. Todas as alterações devem ser feitas nos ficheiros de código e estes devem ser recompilados.

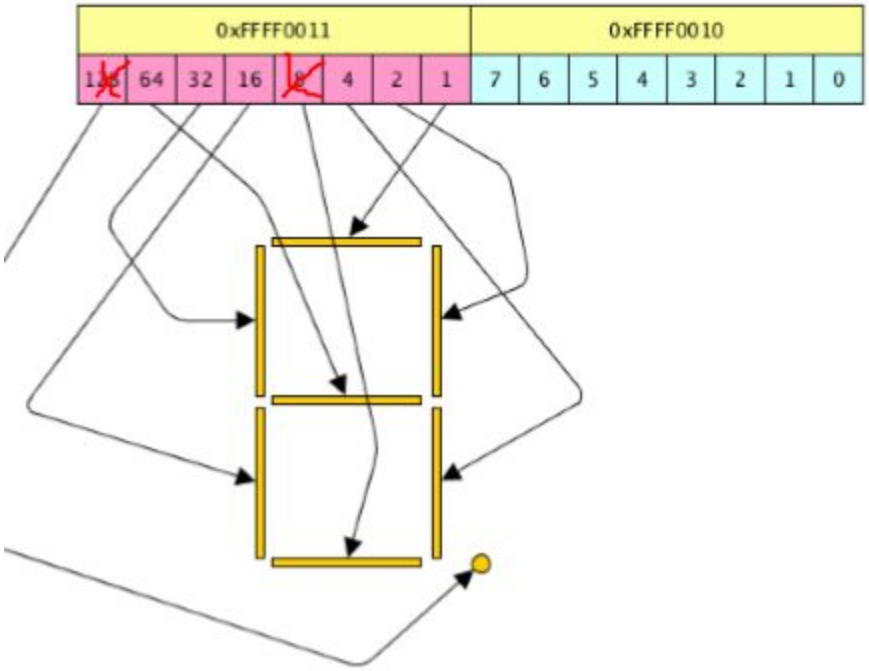
Compilar no gcc com a flag -c produz um ficheiro de objeto (linguagem máquina), com a flag -g adiciona informação extra como os números de linhas para excertos do código fonte de forma a permitir o debugging passo a passo.

A flag -o permite escolher o nome do executável a ser criado quando compilado sem flags adicionais.

O gcc permite executar qualquer etapa da compilação.

13 - B

Ver trabalho prático para informação adicional. 0x77 corresponde a 0b01110111. Portanto são desenhados os segmentos não marcados na figura, resultando no desenho de um A.



14 - B

A = 10 ocorre apenas uma vez, inicializando o ciclo, de seguida é verificada a condição de paragem, é corrido o corpo do ciclo e por fim ocorre o passo iterativo (a--), recomeçando agora o ciclo com a verificação da condição de paragem.

Trabalhando com inteiros, se a < 1 é verdade então a > 0 é falso. Quando a < 1 a instrução “slti $t0, $s0, 1” coloca $t0 a 0, caso contrário $t0 fica a 1. O ciclo deve terminar quando a > 0, ou seja, quando a < 1 e portanto $t0 = 0. Ou seja, se $t0 = 0, devemos saltar para out, excluindo assim as opções A e D.

No final do ciclo queremos decrementar a, portanto fazemos “addiu $s0, $s0, -1” (equivalente a “subiu $s0, $s0, 1”).

15 - B

Memoria principal = $2^{bits\ endereçamento} = 2^{10+10+10} = 2^{30}$ bytes = 1 GB

Memoria cache = $2^{bits\ endereçamento\ -\ tag} = 2^{30-10} = 2^{20}$ bytes = 1 MB

Tamanho bloco = $2^{offset\ bits} = 2^{10}$ = 1KB