Esta ficha foi elaborada para acompanhar o estudo da disciplina de Arquitetura de Computadores I (a2s1) 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. Codifica, em código máquina, as seguintes instruções, considerando que as instruções de tipo R têm código de operação 000000, as instruções jump têm código de operação 000010, e branch on not equal 000101:

a) `add  $2, $8, $3` `(considere-se o código de função 100000)`

A instrução é do tipo R, logo o código de operação é 000000 e o código de função é o indicado em enunciado, por 100000. Estes são os campos `opcode` e `funct`, respetivamente. As instruções do tipo R também contêm outros campos, todos de 5 bits, entre os quais, os registos rs, rt e rd e um campo de designação de deslocamento `shamt`, que não terá qualquer significado neste caso em particular (tendo o valor 00000). Já os campos rs, rt e rd, são os identificadores dos registos em causa. Sendo que a instrução está na ordem `add  $rd, $rs, $rt`, então rs será 8 em binário (01000), rt será 3 em binário (00011) e rd será 2 em binário (00010). O código máquina será então o seguinte:

`000000  01000  00011  00010  00000  100000 = 0x01031020`

b) `addi $2, $8, 100` `(considere-se o código de operação 001000)`

A instrução é do tipo I, logo não temos código de função. Sendo o código de operação é 001000, indicado em enunciado, falta-nos agora indicar o valor dos três restantes campos. As instruções do tipo I contêm outros campos, dois de 5 bits e um de 16 bits, entre os quais, os registos rs, rt e um campo de designação de uma constante, vulgarmente denominada de imediato, representado sempre em complemento para 2, com extensão de sinal para 16 bits. Já os campos rs e rt, são os identificadores dos registos em causa. Sendo que a instrução está na ordem `addi  $rt, $rs, Imm`, então rs será 8 em binário (01000), rt será 2 em binário (00010) e Imm será 100 em complemento para dois, representado em 16 bits (0000 0000 0110 0100). O código máquina será então o seguinte:

`000000  01000  00010  0000000001100100 = 0x01031020`

c) `addu $2, $8, $3` `(considere-se o código de função 100001)`

A instrução é do tipo R, logo o código de operação é 000000 e o código de função é o indicado em enunciado, por 100001. Estes são os campos `opcode` e `funct`, respetivamente. As instruções do tipo R também contêm outros campos, todos de 5 bits, entre os quais, os registos rs, rt e rd e um campo de designação de deslocamento `shamt`, que não terá qualquer significado neste caso em particular (tendo o valor 00000). Já os campos rs, rt e rd, são os identificadores dos registos em causa. Sendo que a instrução está na ordem `addu  $rd, $rs, $rt`, então rs será 8 em binário (01000), rt será 3 em binário (00011) e rd será 2 em binário (00010). O código máquina será então o seguinte:

`000000  01000  00011  00010  00000  100001 = 0x01031021`

d) `sll  $2, $8, 4` `(considere-se o código de função 000000)`

A instrução é do tipo R, logo o código de operação é 000000 e o código de função é o indicado em enunciado, por 000000. Estes são os campos `opcode` e `funct`, respetivamente. As instruções do tipo R também contêm outros campos, todos de 5 bits, entre os quais, os registos rs, rt e rd e um campo de designação de deslocamento `shamt`, que terá o valor 4 (tendo o valor 00100). Já os campos rs, rt e rd, são os identificadores dos registos em causa. Sendo que a instrução está na ordem `sll  $rd, $rt, shift`, então rt será 8 em binário (01000), rd será 2 em binário (00010) e rs será 0 em binário, dado que não tem significado nesta instrução (00000). O código máquina será então o seguinte:

`000000  00000  01000  00010  00100  000000 = 0x00081100`

e) `sw  $2, 200($3)` `(considere-se o código de operação 101011)`

A instrução é do tipo I, logo não temos código de função. Sendo o código de operação é 101011, indicado em enunciado, falta-nos agora indicar o valor dos três restantes campos. As instruções do tipo I contêm outros campos, dois de 5 bits e um de 16 bits, entre os quais, os registos rs, rt e um campo de designação de uma constante, vulgarmente denominada de imediato, representado sempre em complemento para 2, com extensão de sinal para 16 bits. Já os campos rs e rt, são os identificadores dos registos em causa. Sendo que a instrução está na ordem `sw  $rt, Imm($rs)`, então rs será 3 em binário (00011), rt será 2 em binário (00010) e Imm será 200 em complemento para dois, representado em 16 bits (0000 0000 1100 1000). O código máquina será então o seguinte:

`101011  00011  00010  0000000011001000 = 0xAC6200C8`

f) `j  fatorial  (considere-se que fatorial se encontra em 0x4004000C)`

A instrução é do tipo J, logo não temos código de função. Sendo o código de operação é 000010, indicado em enunciado, falta-nos agora indicar o valor do único campo restante - o target, de 26 bits. Para obtermos este valor, primeiro precisamos de considerar que o endereço final de target é constituído pelos 4 bits mais significativos do $pc concatenados com o imediato da instrução, multiplicado por 4. Assim, vejamos os 28 bits menos significativos do endereço target indicado em enunciado e dividamos por 4, isto é, desloquemos para a direita 2 vezes. Assim temos $004000C_{16}$ /4 = 0000 0000 0100 0000 0000 0000 1100_2 / 4 = 0000 0000 0001 0000 0000 0000 0011_2. O código máquina será então o seguinte:

```
000010  00000000010000000000000011 = 0x08010003
```

g) `bne  $2, $8, endif  (considere-se endif salta 5 instruções à frente)`

A instrução é do tipo I, logo não temos código de função. Sendo o código de operação é 000101, indicado em enunciado, falta-nos agora indicar o valor dos três restantes campos. As instruções do tipo I contêm outros campos, dois de 5 bits e um de 16 bits, entre os quais, os registos rs, rt e um campo de designação de uma constante, vulgarmente denominada de imediato, representado sempre em complemento para 2, com extensão de sinal para 16 bits. Já os campos rs e rt, são os identificadores dos registos em causa. Sendo que a instrução está na ordem `bne $rs, $rt, LABEL`, então rs será 2 em binário (00010), rt será 8 em binário (01000) e a LABEL terá correspondência a um deslocamento que será de 5 instruções no sentido positivo, isto é, 0000 0000 0000 0101. O código máquina será então o seguinte:

```
000101  00010  01000  0000000000000101 = 0x14480005
```

h) `jr  $ra  (considere-se o código de função 001000)`

A instrução é do tipo R, logo o código de operação é 000000 e o código de função é o indicado em enunciado, por 001000. Estes são os campos `opcode` e `funct`, respetivamente. As instruções do tipo R também contêm outros campos, todos de 5 bits, entre os quais, os registos rs, rt e rd e um campo de designação de deslocamento `shamt`, que não terá qualquer significado (tendo o valor 00000). Já os campos rs, rt e rd, são os identificadores dos registos em causa. Sendo que a instrução está na ordem `jr  $rs`, então rs será 31 em binário (11111) e rd e rt serão 0 em binário, dado que não têm significado nesta instrução (00000). O código máquina será então o seguinte:

```
000000  11111  00000  00000  00000  001000 = 0x03E00008
```

i) `jal  bubblesort  (considere-se o código de operação 000011 e endereço 0x4004000C)`

A instrução é do tipo J, logo não temos código de função. Sendo o código de operação é 000011, indicado em enunciado, falta-nos agora indicar o valor do único campo restante - o target, de 26 bits. Para obtermos este valor, primeiro precisamos de considerar que o endereço final de target é constituído pelos 4 bits mais significativos do $pc concatenados com o imediato da instrução, multiplicado por 4. Assim, vejamos os 28 bits menos significativos do endereço target indicado em enunciado e dividamos por 4, isto é, desloquemos para a direita 2 vezes. Assim temos $004000C_{16}$ /4 = 0000 0000 0100 0000 0000 0000 1100_2 / 4 = 0000 0000 0001 0000 0000 0000 0011_2. O código máquina será então o seguinte:

```
000011  00000000010000000000000011 = 0x0C010003
```

2. Qual é a diferença, em termos de arquitetura (a nível físico), entre os registos $s e os registos $t?

Entre os registos $s e os registos $t não existe qualquer diferença de arquitetura. Em termos de utilização destes registos - tais como dos restantes $a e $v - é que se torna conveniente usar uma convenção, que identifica registos $s como de salvaguarda de dados, registos $t como temporários, registos $v como retorno de procedimentos ou funções e registos $a como argumentos para as mesmas.

3. Como é que se processa a tradução dos programas em Assembly do MIPS, relativamente a instruções que contém etiquetas (labels), como a instrução jump? Por outras palavras, sabendo que nelas se escreve uma etiqueta, a que corresponde esta e como é que é feita a correspondência entre o seu significado real e a identificação de label?

Quando um programa contém uma instrução jump, o assembler, numa primeira passagem, irá criar uma symbol table, na qual cria uma correspondência direta entre uma etiqueta e um endereço, sempre que lê uma instrução jump ou jump-and-link - dado o endereçamento absoluto (contém o endereço na própria instrução). Isto permite a que em execução, o assembler já consiga reconhecer, com complexidade constante em termos de tempo, a que endereço corresponde uma determinada label.

4. Explica o processamento de toda a instrução jal.

A instrução jump-and-link (jal) tem um comportamento igual à instrução jump (tendo inclusivé o mesmo formato - tipo J) diferindo apenas no facto de preservar o estado do registo $pc num outro registo apropriado - o $ra.

Sendo esta instrução usada aquando da invocação de procedimentos ou funções, sempre que se executa uma nova função, a forma de poder regressar à função que invocou (caller) é através do endereço de instrução presente no registo $pc, aquando da invocação, que contém sempre o endereço da instrução a ser executada de seguida. Assim, quando se executa jump-and-link, o assembler irá guardar em $ra o valor atual de $pc, de forma a que se possa regressar à primeira função.

5. De que tipo é a instrução jump register? Explica como é que a instrução se processa quando se executa, por exemplo, jr $ra.

A instrução jump register (jr) é uma instrução do tipo R, onde apenas o campo respetivo ao registo rs é variável de acordo com o argumento da instrução. Por esta razão, há quem considere que esta instrução é do tipo JR, um tipo que tem o formato das instruções de tipo R, mas onde só o campo rs é que varia. Quando se executa `jr $ra`, o conteúdo do registo $ra deve ser copiado para o registo $pc, sendo este valor um endereço de uma instrução a ser executada.

6. Considerem-se as variáveis f, g, h, i e j estão atribuídas aos registos $s0-$s4, respetivamente. Consideremos também, que os endereços-base dos arrays A e B estão nos registos $s6 e $s7, respetivamente.

a) Como é que se traduz, para linguagem Assembly do MIPS, a seguinte operação em C?

f = g - A[B[4]];

```
lw      $s0, 16($s7)          # f = B[4]
sll     $s0, $s0, 2           # f = f * 4 = B[4] * 4
add     $s0, $s0, $s6         # f = &A[f] = &A[B[4]]
lw      $s0, 0($s0)           # f = A[f] = A[B[4]]
sub     $s0, $s1, $s0         # f = g - f = g - A[B[4]]
```

b) A que corresponde a instrução "`lw  $s0, 4($s6)`", em C?

Em C, a instrução corresponde a `f = A[1]`.

7. No simulador MARS estão incluídas duas instruções distintas para o cálculo da divisão entre registos (referimo-nos à instruções de `div`). Sendo uma destas uma pseudo-instrução, a qual recebe como registos de escrita e leitura os registos rs, rt e rd, como é que podemos implementar a divisão em instruções nativas? A pseudo-instrução em causa é a seguinte:

```
    div     $s0, $s1, $s2         # grava em $s0 o valor de $s1/$s2
```

```
div     $s1, $s2              # divisão de $s1/$s2, onde o resultado vai para $HI e $LO
mflo    $s0                   # receber 32 bits menos significativos da divisão em $s0
```

8. Qual a importância da salvaguarda de valores após a invocação de novas funções? É importante guardar o conteúdo de determinados registos sempre? Se não, em que casos é que é necessário e porquê?

Quando um programa contém uma instrução de jump-and-link, isto é, quando invoca uma função, esta função em causa pode ser terminal ou não-terminal (leaf ou non-leaf). Uma função terminal (ou leaf) é uma função que não invoca qualquer outra função no seu decurso de execução. Por outro lado, uma função não-terminal (ou non-leaf) é uma função que invoca outras funções ao longo da sua execução.

A salvaguarda de valores dos registos $s é feita apenas em funções não-terminais, que precisam de usar os registos $s para que, após a invocação de uma nova função (ou da mesma, se for o caso de recursividade), nenhum valor se perca. No caso das funções terminais só se guardam os valores dos registos $s caso a utilização de registos exceda os disponíveis exceto estes. Não é necessário salvaguardar os valores dos registos $s em funções terminais pelo facto de que, se se seguir a convenção, não haverá qualquer risco de perda de qualquer valor.

9. Escreve em linguagem Assembly do MIPS, respeitando as convenções de utilização dos registos, uma função `int impar(int x)`, que retorna o valor `1` caso o argumento x seja ímpar, e 0 caso o argumento x seja par.

```
int impar(int x) {
  if (x % 2 == 0)
    return 0;
  else
    return 1;
}
```

```
impar:  rem     $t0, $t0, 2           # ($t0) = x % 2
        bnez    $t0, else             # if (x % 2 == 0)
        li      $v0, 0                # return 0
        jr      $ra
else:   li      $v0, 1                # return 1
        jr      $ra
```

10. Escreve em linguagem Assembly do MIPS a seguinte função `int bitCount(unsigned x)`, que conta o número de bits que se encontram a 1 ao longo de uma sequência de bits. Segue as convenções do MIPS, para a realização deste exercício.

```
int bitCount(unsigned x) {
  int bit;
  if (x == 0)
    return 0;
  bit = x & 0x1;
  return bit + bitCount(x >> 1);
}
```

```
bitCount:       subi    $sp, $sp, 8           # libertar 8 bytes na stack
                sw      $ra, 0($sp)           # guardar valor de $ra nos primeiros 4 bytes
                sw      $s0, 4($sp)           # guardar valor de #s0 nos segundos 4 bytes
if:             bnez    $a0, else             # if (x == 0)
                li      $v0, 0                # return 0
                lw      $ra, 0($sp)           # restaurar valor de $ra
                lw      $s0, 4($sp)           # restaurar valor de $s0
                addi    $sp, $sp, 8           # restaurar espaço na stack
                jr      $ra
else:           andi    $s0, $a0, 1           # bit = x & 0x1
                srl     $a0, $a0, 1           # ($a0) = x >> 1
                jal     bitCount              # bitCount(x>>1)
                lw      $ra, 0($sp)           # restaurar valor de $ra
                lw      $s0, 4($sp)           # restaurar valor de $s0
                addi    $sp, $sp, 8           # restaurar espaço na stack
                jr      $ra
```