Prova P1 **Disciplina: Organização de Computadores**

Professor: Ney Laert Vilar Calazans

Aluno: 19/abril/2017

Lista de associação de números e mnemônicos para os registradores do MIPS

Número (Decimal)	Nome
0	$zero
1	$at
2	$v0
3	$v1
4	$a0
5	$a1
6	$a2
7	$a3
8	$t0
9	$t1
10	$t2
11	$t3
12	$t4
13	$t5
14	$t6
15	$t7

Número (Decimal)	Nome
16	$s0
17	$s1
18	$s2
19	$s3
20	$s4
21	$s5
22	$s6
23	$s7
24	$t8
25	$t9
26	$k0
27	$k1
28	$gp
29	$sp
30	$fp
31	$ra

1. (3,0 pontos) Montagem/Desmontagem de código objeto. Abaixo se mostra uma listagem gerada pelo ambiente MARS como resultado da montagem de um programa. Pede-se que se substituam as triplas interrogações pelo texto que deveria estar em seu lugar (existem 6 triplas ???). Em alguns casos, isto implica gerar código objeto, e/ou gerar código intermediário e/ou gerar código fonte. Caso uma instrução a ser colocada no lugar das interrogações seja um salto, expresse na área do código fonte/intermediário o exato endereço para onde ela salta (em hexa e/ou com o rótulo associado à linha).
   Dica 1: Dêem muita atenção ao tratamento de endereços e rótulos.
   Dica 2: Tomem muito cuidado com a mistura de representações numéricas: hexa, binário, complemento de 2, etc.
   Obrigatório: Mostre os desenvolvimentos para obter os resultados, justificando.

```
        Endereço   Cód.Objeto  Código Intermediário          Código Fonte
[1]     0x00400000 0x3c011001 lui $1,0x00001001      3    main:  la     $a0,d1
[2]     0x00400004 0x34240000 ori $4,$1,0x00000000
[3]     0x00400008 0x3c011001 lui $1,0x00001001      4           la     $a1,d2
[4]     0x0040000c 0x34250004 ori $5,$1,0x00000004
[5]     0x00400010 0x0c10000b jal 0x0040002c         5           jal    SSH
[6]     0x00400014 0x3c011001 lui $1,0x00001001      6           la     $t0,sshd
[7]     0x00400018 0x34280008 ori $8,$1,0x00000008
[8]     0x0040001c 0xad020000 sw $2,0x00000000($8)   7           sw     $v0,0($t0)
[9]     0x00400020 0xad030004      ???               8                ???
[10]    0x00400024 0x2402000a addiu $2,$0,0x0000000a 9           li     $v0,10
[11]    0x00400028 0x0000000c syscall                10          syscall
[12]    0x0040002c 0x340affff ori $10,$0,0x0000ffff  11   SSH:   ori    $t2,$zero,0xFFFF
[13]    0x00400030 0x8c880000 lw $8,0x00000000($4)   12          lw     $t0,0($a0)
[14]    0x00400034 0x010a4024 and $8,$8,$10          13          and    $t0,$t0,$t2
[15]    0x00400038 0x8ca90000 lw $9,0x00000000($5)   14          lw     $t1,0($a1)
[16]    0x0040003c      ???        ???               15          and    $t1,$t1,$t2
[17]    0x00400040 0x01091021 addu $2,$8,$9          16          addu   $v0,$t0,$t1
[18]    0x00400044 0x3c0a0001 lui $10,0x00000001     17          lui    $t2,1
[19]    0x00400048 0x01425024 and $10,$10,$2         18          and    $t2,$t2,$v0
[20]    0x0040004c      ???        ???               19          beq    $t2,$zero,vue0
[21]    0x00400050 0x24030001 addiu $3,$0,0x00000001 20          addiu  $v1,$zero,1
[22]    0x00400054 0x3c0a0000 lui $10,0x00000000     21          lui    $t2,0
[23]    0x00400058 0x354affff ori $10,$10,0x0000ffff 22          ori    $t2,$t2,0xFFFF
[24]    0x0040005c 0x004a1024 and $2,$2,$10          23          and    $v0,$v0,$t2
[25]    0x00400060 0x03e00008 jr $31                 24          jr     $ra
[26]    0x00400064 0x00001821 addu $3,$0,$0          25   vue0:  addu   $v1,$zero,$zero
[27]    0x00400068 0x03e00008 jr $31                 26          jr     $ra
```

2. (4,0 pontos) O programa em linguagem de montagem do MIPS abaixo faz um processamento bem específico. (a) Descreva em uma frase o que ele faz. (b) Aponte no código fonte **todas** as pseudo-instruções que nele existem. (c) Diga o que acontece com a área de memória que contém os dados do programa após a execução do mesmo, especificando se algo é escrito na memória, onde e que valor(es) é(são) escrito(s). (d) Comente o programa semanticamente.
Dica: Nas linhas 6, 8, 12 e 15 constantes imediatas são especificadas como caracteres ASCIIE, o que é aceito pelo montador MARS.

```
1         .text
2         .globl loop
3    main: la     $t0,axiom
4    loop: lbu    $t1,0($t0)
5          beq    $t1,$zero,end
6          xori   $t2,$t1,'G'
7          beq    $t2,$zero,chG
8          xori   $t2,$t1,'I'
9          beq    $t2,$zero,chI
10   next: addiu  $t0,$t0,1
11         j      loop
12   chG:  addiu  $t3,$zero,'I'
13         sb     $t3,0($t0)
14         j      next
15   chI:  addiu  $t3,$zero,'G'
16         sb     $t3,0($t0)
17         j      next
18   end:  li     $v0,10
19         syscall
20         .data
21   axiom:.asciiz "G is better than I!"
```

3. (3,0 pontos) Verdadeiro ou Falso. Abaixo aparecem 5 afirmativas. Marque com V as afirmativas verdadeiras e com F as falsas. Se não souber a resposta correta, deixe em branco, pois cada resposta correta vale 0,6 pontos, mas cada resposta incorreta desconta 0,3 pontos do total positivo de pontos. Não é possível que a questão produza uma nota menor do que 0 pontos.

a) ( ) Um processador de 16 bits possui um barramento de endereços para se comunicar com a memória de 24 fios e um registrador program counter (PC) com o mesmo número de bits, 24. Assuma que o processador emprega um modelo de memória que usa endereçamento a byte. Logo, o mapa de memória acessível aos programas deste processador é de 32Mbytes.

b) ( ) Suponha que o registrador **$t1** contém o valor **0xFA45778D**. Nesta situação, após executar a instrução **xori $t1,$t1,0x8000**, o conteúdo do registrador **$t1** passará a ser **0xFA45F78D**.

c) ( ) Suponha que ao executar uma instrução **lh** que leu dados a partir do endereço de memória **0x10010018**, escreveu-se no registrador **$t0** o número **0xFFFF83A4**. Sabendo que a implementação do MIPS onde a instrução foi executada é *little endian*, os valores que estão armazenados nos endereços de memória **0x10010018**, **0x10010019**, são, respectivamente **0xA4**, e **0x83**. Não se sabe a partir das informações fornecidas o que está contido nos endereços de memória **0x1001001A** e **0x1001001B**.

d) ( ) Nas instruções **lw** e **sw** o primeiro operando especificado no código fonte é sempre um registrador usado como destino da instrução (aquele onde a instrução escreve algum valor) e nunca como registrador fonte.

e) ( ) Após executar a instrução **addi $t0,$zero,0x85AB**, o conteúdo do registrador **$t0** será **0xFFFF85AB**, independente do seu valor inicial.

Prova P1 Disciplina: Organização de Computadores

Professor: Ney Laert Vilar Calazans

Aluno: 19/abril/2017

Gabarito

Lista de associação de números e mnemônicos para os registradores do MIPS

Número (Decimal)	Nome
0	$zero
1	$at
2	$v0
3	$v1
4	$a0
5	$a1
6	$a2
7	$a3
8	$t0
9	$t1
10	$t2
11	$t3
12	$t4
13	$t5
14	$t6
15	$t7

Número (Decimal)	Nome
16	$s0
17	$s1
18	$s2
19	$s3
20	$s4
21	$s5
22	$s6
23	$s7
24	$t8
25	$t9
26	$k0
27	$k1
28	$gp
29	$sp
30	$fp
31	$ra

1. (3,0 pontos) Montagem/Desmontagem de código objeto. Abaixo se mostra uma listagem gerada pelo ambiente MARS como resultado da montagem de um programa. Pede-se que se substituam as triplas interrogações pelo texto que deveria estar em seu lugar (existem 6 triplas ???). Em alguns casos, isto implica gerar código objeto, e/ou gerar código intermediário e/ou gerar código fonte. Caso uma instrução a ser colocada no lugar das interrogações seja um salto, expresse na área do código fonte/intermediário o exato endereço para onde ela salta (em hexa e/ou com o rótulo associado à linha).
   Dica 1: Dêem muita atenção ao tratamento de endereços e rótulos.
   Dica 2: Tomem muito cuidado com a mistura de representações numéricas: hexa, binário, complemento de 2, etc.
   Obrigatório: Mostre os desenvolvimentos para obter os resultados, justificando.

```
        Endereço   Cód.Objeto  Código Intermediário          Código Fonte
[1]     0x00400000 0x3c011001 lui $1,0x00001001       3     main: la     $a0,d1
[2]     0x00400004 0x34240000 ori $4,$1,0x00000000
[3]     0x00400008 0x3c011001 lui $1,0x00001001       4           la     $a1,d2
[4]     0x0040000c 0x34250004 ori $5,$1,0x00000004
[5]     0x00400010 0x0c10000b jal 0x0040002c          5           jal    SSH
[6]     0x00400014 0x3c011001 lui $1,0x00001001       6           la     $t0,sshd
[7]     0x00400018 0x34280008 ori $8,$1,0x00000008
[8]     0x0040001c 0xad020000 sw $2,0x00000000($8)    7           sw     $v0,0($t0)
[9]     0x00400020 0xad030004      ???                 8                ???
[10]    0x00400024 0x2402000a addiu $2,$0,0x0000000a  9           li     $v0,10
[11]    0x00400028 0x0000000c syscall                 10          syscall
[12]    0x0040002c 0x340affff ori $10,$0,0x0000ffff   11    SSH:  ori    $t2,$zero,0xFFFF
[13]    0x00400030 0x8c880000 lw $8,0x00000000($4)    12          lw     $t0,0($a0)
[14]    0x00400034 0x010a4024 and $8,$8,$10           13          and    $t0,$t0,$t2
[15]    0x00400038 0x8ca90000 lw $9,0x00000000($5)    14          lw     $t1,0($a1)
[16]    0x0040003c      ???        ???                15          and    $t1,$t1,$t2
[17]    0x00400040 0x01091021 addu $2,$8,$9           16          addu   $v0,$t0,$t1
[18]    0x00400044 0x3c0a0001 lui $10,0x00000001      17          lui    $t2,1
[19]    0x00400048 0x01425024 and $10,$10,$2          18          and    $t2,$t2,$v0
[20]    0x0040004c      ???        ???                19          beq    $t2,$zero,vue0
[21]    0x00400050 0x24030001 addiu $3,$0,0x00000001  20          addiu  $v1,$zero,1
[22]    0x00400054 0x3c0a0000 lui $10,0x00000000      21          lui    $t2,0
[23]    0x00400058 0x354affff ori $10,$10,0x0000ffff  22          ori    $t2,$t2,0xFFFF
[24]    0x0040005c 0x004a1024 and $2,$2,$10           23          and    $v0,$v0,$t2
[25]    0x00400060 0x03e00008 jr $31                  24          jr     $ra
[26]    0x00400064 0x00001821 addu $3,$0,$0           25    vue0: addu   $v1,$zero,$zero
[27]    0x00400068 0x03e00008 jr $31                  26          jr     $ra
```

Solução da Questão 1 (3,0 pontos). Cada ??? vale 0,5 pontos

```
[9]     0x00400020 0xad030004      ???                 8                ???
```

O que se quer aqui é gerar os códigos intermediário e fonte de uma instrução, dado apenas seu código objeto, ou seja, uma operação de desmontagem de código. Para realizar a desmontagem, entra-se com os 6 primeiros bits do código objeto (101011, ou 43) na Tabela da Figura A.10.2 do Apêndice A, o que identifica a instrução como sendo um **sw**. Com esta descoberta, usa-se novamente o Apêndice A para dele extrair o formato da instrução, qual seja:

```
                    sw   rt,offset(rs)
                    0x2b  rs rt  offset
Número de bits/campo:     6   5  5     16
```

A geração do código intermediário a partir daqui é direta, basta extrair os bits dos campos do formato, obter o número dos registradores e o valor da constante imediata. Pode-se usar a tabela no início da prova para relacionar o número dos registradores com seu nome e daí gerar o código fonte. Assim, tem-se **rs**=01000 (8, bits 25-21 do código objeto, correspondendo ao registrador **$t0**), **rt**=00011 (3, bits 20-16 do código objeto, correspondendo ao registrador **$v1**) e **offset**=0x0004=4.

Resposta Final:

```
[9]    0x00400020  0xad030004     sw  $3,0x0004($8)   8          sw  $v1,4($t0)

[16]   0x0040003c      ???           ???              15         and   $t1,$t1,$t2
```

O código fonte contém uma instrução (and $t1,$t1,$t2). Assim, basta gerar os códigos intermediário e objeto desta instrução. O formato da instrução **and**, retirado do Apêndice A é:

```
                    and   rd,rs,rt      : ling. de montagem
                    0x0 rs rt rd 0 0x24 : cód. objeto
Número de bits/campo:     6  5  5  5  5  6  :
```

O código intermediário, neste caso, é trivialmente formado a partir do código fonte, apenas substituindo os nomes dos registradores pelos seus números respectivos em decimal, o que fornece `and $9,$9,$10`. O código objeto em 32 bits é facilmente gerado, **apenas atentando** para a ordem de apresentação dos registradores, que é diferente nos formatos fonte (ling. de montagem) e objeto. Concatena-se 000000 (0 em 6 bits) com o endereço do **rs** no banco, 01001 (9 em binário 5 bits código do registrador $9 ou $t1), com o endereço do **rt** no banco, 01010 ($10=$t2, ou 10 em 5 bits), com o o endereço do **rd** no banco, 01001 ($9=$t1 ou 2 em 5 bits), com 00000 (constante 0 em 5 bits) e com 100100 (correspondendo a 0x24 em 6 bits). O resultado é então 0000 0001 0010 1010 0100 1000 0010 0100. Convertendo este código de 32 bits em hexadecimal, tem-se o formato final do código objeto, o que faltava para a solução desta questão: 0x012A4824.

Resposta final:

```
[16]   0x0040003c  0x012A4824    and $9,$9,$10       15         and   $t1,$t1,$t2

[20]   0x0040004c      ???          ???              19         beq   $t2,$zero,vue0
```

Novamente, se deseja gerar os códigos intermediário e objeto desta instrução. O formato da instrução **beq**, retirado do Apêndice A é:

```
                    beq   rs,rt,label   : ling. de montagem
                    0x4 rs rt offset    : cód. objeto
Número de bits/campo:     6  5  5    16      :
```

Em primeiro lugar, note-se que o rótulo (em inglês, *label*) mencionado na instrução beq, vue0, encontra-se na linha [26] do programa, enquanto que o beq em si encontra-se na linha [20]. Como entre estas linhas existem apenas instruções, o offset é determinado pelo número de linhas que existe entre a linha seguinte ao beq (linha [21]) e a linha do rótulo destino do salto condicional (linha [26]). Logo o offset é 5 (em decimal, ou o mesmo numeral em hexadecimal). Seguindo-se o formato da instrução, gera-se agora facilmente o código objeto, concatenando 000100 (o opcode 0x4 expresso em binário, 6 bits) com o código do registrador rs em 5 bits (01010, código do registrador $t2=$10 em binário 5 bits), com o código do registrador rt em 5 bits (00000, código do registrador $zero=$0 em binário 5 bits), com o código do offset expresso em 16 bits (0000 0000 0000 0101, ou 5, expresso em binário 16 bits). O resultado é 0001 0001 0100 0000 0000 0000 0000 0101, que expresso em hexadecimal fica 0x11400005.

Resposta Final:

```
[20]   0x0040004c  0x11400005    beq   $10,$0,0x0005   19         beq   $t2,$zero,vue0
```

Fim da Solução da Questão 1 (3,0 pontos)

2. (4,0 pontos) O programa em linguagem de montagem do MIPS abaixo faz um processamento bem específico. (a) Descreva em uma frase o que ele faz. (b) Aponte no código fonte **todas** as pseudo-instruções que nele existem. (c) Diga o que acontece com a área de memória que

contém os dados do programa após a execução do mesmo, especificando se algo é escrito na memória, onde e que valor(es) é(são) escrito(s). (d) Comente o programa semanticamente.
Dica: Nas linhas 6, 8, 12 e 15 constantes imediatas são especificadas como caracteres ASCIIE, o que é aceito pelo montador MARS.

```
1          .text
2          .globl loop
3    main: la     $t0,axiom          # $t0 contém um ponteiro para a cadeia axiom
4    loop: lbu    $t1,0($t0)         # $t1 recebe o próximo caracter da cadeia
5          beq    $t1,$zero,end      # Salta p/ fim se caracter lido é o último (NULL)
6          xori   $t2,$t1,'G'        # Compara para ver se caracter lido é G
7          beq    $t2,$zero,chG      #      se for G, salta para trocar por I
8          xori   $t2,$t1,'I'        # Senão, compara para ver se caracter lido é I
9          beq    $t2,$zero,chI      #      se for I, salta pata trocar por G
10   next: addiu  $t0,$t0,1          # Não sendo G nem I ou depois da troca, avança
                                          ponteiro para o próximo caracter
11         j      loop               # E volta para testar próximo caracter
12   chG:  addiu  $t3,$zero,'I'      # Troca de G por I: gera caracter I
13         sb     $t3,0($t0)         # Escreve em axiom, usa ponteiro p/ carac. atual
14         j      next               # E continua com o processamento da cadeia
15   chI:  addiu  $t3,$zero,'G'      # Troca de I por G: gera caracter G
16         sb     $t3,0($t0)         # Escreve em axiom, usa ponteiro p/ carac. atual
17         j      next               # E continua com o processamento da cadeia
18   end:  li     $v0,10             # Fim do processamento, prepara saída do programa
19         syscall                   # E sai.
20         .data
21   axiom:.asciiz "G is better than I!"    # Cadeia a processar
```

Solução da Questão 2 (4,0 pontos)

a) Este programa substitui todas as instâncias de caracteres `G´ e `I´ na cadeia axiom respectivamente pelos caracteres `I´ e `G´.
b) As duas linhas que contêm pseudo-instruções estão salientadas no código acima (linhas 3 e 18). Os mnemônicos la e li sempre são pseudo-instruções.
c) A área de dados do programa é alterada nas posições da cadeia **axiom** onde existem caracteres `G´ e `I´ (primeira e penúltima posição da cadeia). A cadeia **axiom** ao final do processamento contém "I is better than G!".
d) Ver código comentado em vermelho acima.

Fim da Solução da Questão 2 (4,0 pontos)

3. (3,0 pontos) Verdadeiro ou Falso. Abaixo aparecem 5 afirmativas. Marque com V as afirmativas verdadeiras e com F as falsas. Se não souber a resposta correta, deixe em branco, pois cada resposta correta vale 0,6 pontos, mas cada resposta incorreta desconta 0,3 pontos do total positivo de pontos. Não é possível que a questão produza uma nota menor do que 0 pontos.

a) ( ) Um processador de 16 bits possui um barramento de endereços para se comunicar com a memória de 24 fios e um registrador program counter (PC) com o mesmo número de bits, 24. Assuma que o processador emprega um modelo de memória que usa endereçamento a byte. Logo, o mapa de memória acessível aos programas deste processador é de 32Mbytes.
b) ( ) Suponha que o registrador **$t1** contém o valor **0xFA45778D**. Nesta situação, após executar a instrução **xori $t1,$t1,0x8000**, o conteúdo do registrador **$t1** passará a ser **0xFA45F78D**.
c) ( ) Suponha que ao executar uma instrução **lh** que leu dados a partir do endereço de memória **0x10010018**, escreveu-se no registrador **$t0** o número **0xFFFF83A4**. Sabendo que a implementação do MIPS onde a instrução foi executada é *little endian*, os valores que estão armazenados nos endereços de memória **0x10010018**, **0x10010019**, são, respectivamente **0xA4**, e **0x83**. Não se sabe a partir das informações fornecidas o que está contido nos endereços de memória **0x1001001A** e **0x1001001B**.
d) ( ) Nas instruções **lw** e **sw** o primeiro operando especificado no código fonte é sempre um registrador usado como destino da instrução (aquele onde a instrução escreve algum valor) e nunca como registrador fonte.
e) ( ) Após executar a instrução **addi $t0,$zero,0x85AB**, o conteúdo do registrador **$t0** será **0xFFFF85AB**, independente do seu valor inicial.

Solução da Questão 3 (3,0 pontos)

a) (**F**) Um processador de 16 bits possui um barramento de endereços para se comunicar com a memória de 24 fios e um registrador program counter (PC) com o mesmo número de bits, 24. Assuma que o processador emprega um modelo de memória que usa endereçamento a byte. Logo, o mapa de memória acessível aos programas deste processador é de 32Mbytes.

Por definição endereçamento a byte implica que cada endereço distinto corresponde ao endereço de 1 byte. Com 24 bits o número de endereços distintos que se pode gerar é exatamente 2^{24}, ou seja 16Mega endereços, cada um correspondendo a um byte de armazenamento em memória. O mapa de memória acessível a este processador então é de apenas 16Mbytes. Logo, a afirmativa é Falsa. **F**

b) (**V**) Suponha que o registrador **$t1** contém o valor **0xFA45778D**. Nesta situação, após executar a instrução **xori $t1,$t1,0x8000**, o conteúdo do registrador **$t1** passará a ser **0xFA45F78D**.

A operação xori realiza o Ou Exclusivo de dois valores de 32 bits bit a bit. Um dos valores é simplesmente o conteúdo do registrador $t1 (o segundo operando da instrução). Conforme a questão assume este valor é 0xFA45778D, que em binário equivale ao numeral expresso por 1111 1010 0100 0101 0111 0111 1000 1101. O segundo valor é obtido do terceiro operando de 16 bits, realizando a extensão de zeros do dito valor (conforme a descrição da instrução xori no Apêndice A). Como o operando vale 0x8000, o valor de 32 bits obtido após a extensão de 0 é 0000 0000 0000 1000 0000 0000 0000. O Ou Exclusivo destes dois vetores binários vai gerar um novo vetor binário onde cada bit será 0 sempre que os dois bits da mesma posição nos dois vetores forem iguais e será 1 caso contrário. Ou seja, cada bit do resultado vai ser igual ao bit do primeiro vetor, exceto no bit 15, onde o inverso do valor do primeiro vetor será obtido. O xor resulta em 1111 1010 0100 0101 1111 0111 1000 1101, ou seja, 0xFA45F78D em hexadecimal. Logo, a afirmativa é Verdadeira. **V**

c) (**V**) Suponha que ao executar uma instrução **lh** que leu dados a partir do endereço de memória **0x10010018**, escreveu-se no registrador **$t0** o número **0xFFFF83A4**. Sabendo que a implementação do MIPS onde a instrução foi executada é *little endian*, os valores que estão armazenados nos endereços de memória **0x10010018**, **0x10010019**, são, respectivamente **0xA4**, e **0x83**. Não se sabe a partir das informações fornecidas o que está contido nos endereços de memória **0x1001001A** e **0x1001001B**.

A instrução lh lê meia palavra (*half word*, 16 bits) da memória, faz a extensão de sinal dos 16 bits lidos para transformar estes em um valor de 32 bits e o resultado é escrito no registrador destino da instrução lh. Como o MIPS emprega um modelo de memória com endereçamento a byte, estes 16 bits ou 2 bytes ocupam dois endereços consecutivos na memória. Começando no endereço 0x10010018 as posições lidas são duas, 0x10010018 e 0x10010019. Como a arquitetura é little endian, o byte menos significativo dos 16 bits é o que reside no endereço menor e o segundo byte lido é o mais significativo. Logo, se $t0 foi escrito com 0xFFFF83A4, o byte menos significativo (0xA4) certamente veio do endereço 0x10010018 e o segundo byte (0x83) veio do endereço seguinte, 0x10010019. A extensão de sinal copia o bit 15 da informação lida da memória (o mais significativo dos 16 lidos). Como este é o bit mais significativo do dígito hexadecimal 8 (ou 1000 em binário) o bit 15 vale 1 e ele é copiado 16 vezes à direita do bit 15, gerando 1111 1111 1111 1111 1000 0011 1010 0100 ou 0xFFFF83A4 em hexadecimal. A última frase da questão também é correta, pois as posições de memória 0x1001001A e 0x1001001B não são acessadas de forma alguma pela instrução lh. Logo, a afirmativa é Verdadeira. **V**

d) (**F**) Nas instruções **lw** e **sw** o primeiro operando especificado no código fonte é sempre um registrador usado como destino da instrução (aquele onde a instrução escreve algum valor) e nunca como registrador fonte.

As instrução lw e sw realizam a leitura de uma palavra (4 bytes ou 32 bits) da memória (instrução lw) e a escrita de uma palavra na memória (instrução sw). O primeiro operando da instrução lw é o registrador que será escrito com a palavra vinda da memória, logo ele é o destino da instrução. No entanto, o primeiro operando da instrução sw é o registrador que contém o dado a ser escrito na memória. Logo, a afirmativa é Falsa. **F**

e) (**V**) Após executar a instrução **addi $t0,$zero,0x85AB**, o conteúdo do registrador **$t0** será **0xFFFF85AB**, independente do seu valor inicial.

A instrução addi é uma instrução aritmética que possui um operando imediato de 16 bits. Este valor, como em toda instrução aritmética é transformado em um valor de 32 bits usando a operação de extensão de sinal. Como o valor imediato é 0x85AB ou 1000 0101 1010 1011, o bit mais significativo vale 1 e a operação de extensão de sinal copiará este bit em 1

dezesseis vezes à esquerda para produzir o valor de 32 bits a ser somado ao registrador especificado pelo segundo operando da instrução ($zero). Este valor então será assim em binário 1111 1111 1111 1111 1000 0101 1010 1011, ou 0xFFFF85AB. Este valor será somado ao conteúdo de $zero que é sempre a constante 0 em 32 bits, resultando no mesmo valor, 0xFFFF85AB, que será armazenado no registrador $t0. Para alcançar este resultado, o valor inicial de $t0 é irrelevante. Logo, a afirmativa é Verdadeira. **V**

Fim da Solução da Questão 3 (3,0 pontos)