Universidade Federal da Fronteira Sul
Curso de Ciência da Computação
Organização de Computadores

1. Considere o seguinte trecho de programa em linguagem Assembly do RISC-V:

```
            .data       0x10010000     # segmento de dados
palavra1:   .word       13
palavra2:   .word       0x15
```

Indique, em hexadecimal, quais os valores dos seguintes rótulos:

palavra1: 0x10010000
palavra2: 0x10010004

O valor armazenado nas posições de memória (13 e 0x15), ou os seus equivalentes em hexadecimal/decimal não são a resposta correta pois a pergunta se refere ao valor dos rótulos, isto é, os endereços de memória que eles armazenam/representam.

2. Considere o seguinte programa em Assembly do RISC-V:

```
            .data       0x10010000     # segmento de dados
var1:       .word       30
var2:       .word       -50
txt:        .string     "Organizacao de Computadores"

            .text       0x400000
main:
      la   s0, var1
      lw   t3, 0(s0)
      la   s1, var2
      lw   t4, 0(s1)
      sw   t3, 0(s1)
      sw   t4, 0(s0)
      and t1, t1, zero
      and t0, t0, zero
      beq t3, zero, haz
      ori t0, zero, 1
haz:
      slt t1, t4, t3
wrt:  li   t0, 111
      la   t2, txt
      sb   t0, 0(t2)
      li   t0, 99
      sb   t0, 15(t2)
```

a) Indique, em hexadecimal, quais os valores dos seguintes rótulos sabendo que a pseudo-instrução `'la'` utiliza 2 instruções de máquina:

var1: 0x10010000
var2: 0x10010004
txt: 0x10010008
main: 0x400000
haz: 0x400030
wrt: 0x400034

b) Qual o valor armazenado nas posições de memória indicadas abaixo ao final da execução do programa:

```
var1: -50
var2: 30
txt: "organizacao de computadores"
```

c) Qual o valor que será atribuído ao campo 'Imm' da instrução 'beq' em função do rótulo 'haz' utilizado na instrução?

Como a instrução do rótulo 'haz' está 8 posições de memória para frente a constante a ser presente na instrução será o valor 4, pois o bit menos significativo da constante não é utilizado.

3. Considere o seguinte programa em Assembly do RISC-V:

```
            .data       0x10010000      # segmento de dados
byte1:      .byte       30
var1:       .space      2
value1:     .word       10
byte2:      .byte       12
value2:     .word       20
```

a) Quantos bytes foram reservados para o rótulo 'var1'? 2 bytes

b) O que acontece se o processador executar a instrução 'lw t0, 0(s0)', considerando que s0 possui o endereço de 'value1'? Qual o motivo? Como resolver essa situação?

Dá erro de acesso a memória. Devido ao desalinhamento da memória, pois na instrução lw o endereço de memória acessado deve ser múltiplo de 4. Resolve-se a situação colocando a diretiva '.align' antes da definição do rótulo 'value1'.

c) A situação presente no item b também acontece ao executar a instrução 'lw t1, 0(s1)', considerando que s1 possui o endereço de 'value2'?

Considerando o programa como se encontra originalmente não haverá erro pois 'value2' está armazenado em uma posição de memória múltipla de 4.

4) Descreva o que acontece com o fluxo de execução do programa e com os registradores durante a chamada das instruções CALL e RET em um programa usando o Assembly do RISC-V.

No RISC-V, as instruções CALL (ou JAL - Jump and Link) e RET (ou JALR - Jump and Link Register) controlam o fluxo de execução do programa realizando a chamada de uma função (CALL) e o retorno da chamada da função (RET). CALL salva o endereço do retorno no registrador ra (ou x1) e salta para o endereço da função (fazendo PC <- PC + IMM). RET retorna ao endereço salvo em ra durante a execução do CALL.

Detalhes da chamada (CALL/JAL):

a) Salvar o ponto de retorno:

A instrução CALL (ou JAL) grava o endereço da instrução seguinte (PC+4) no registrador ra (ou x1). Este registrador armazena o endereço de onde a execução deve retornar quando a função terminar.

b) Salto para a função:

A instrução CALL salta para o endereço da função que está sendo chamada, mudando o fluxo de execução (PC -< PC + 4).

Detalhes do retorno (RET/JALR):

a) Restaurar o fluxo:

A instrução RET (ou JALR) lê o endereço salvo no registrador ra (ou x1) e coloca no PC (PC <- ra).

b) Salto de volta:

A CPU salta para o endereço salvo no registrador ra, restaurando o fluxo de execução para a instrução seguinte à chamada da função.

Em resumo: CALL salva o ponto de retorno em ra e salta para a função, enquanto RET usa o valor em ra para retornar ao ponto de chamada.

5) Complete o código abaixo. O código é a parte inicial de um programa com Assembly do RISC-V que percorre o vetor de bytes `'vetor'` e conta o número de bytes cujo valor é igual a 1 e armazena o resultado da contagem na variável `'conta'`. (Pode assumir que a dimensão do vetor sala é sempre 64 bytes).

```
            .data       # segmento de dados
vetor:      .space      64
conta:      .word       0

            .text
main:
      li t1, 0    # contador do número de '1's
      li t2, 0    # controle do laço
      li t3, 1    # constante 1
     li t4, 64
      la t5, vetor
      lb t0, 0($t5)
laco:
      beq t2, t4,fim
      bne t0, t3, continua
      addi t1, t1, 1
continua:
      addi t2, t2, 1
      addi t5, t5, 1

      j laco
fim: la t2, conta
      sw t1, 0(t2)
```

6. Considere que já se encontra implementado a função `randnum`, a qual devolve um inteiro aleatório entre 0 e um dado número limite:

- argumento: número máximo aleatório pretendido (em a0)
- resultados: inteiro aleatório entre 0 e argumento a0 menos 1

Escreva uma função chamada `Joga`, usando o Assembly do RISC-V, que recebe como argumento um inteiro em a0 e chama a função `randnum` com o argumento 10. Caso o resultado da chamada de `randnum` seja igual ao argumento em a0, a função `Joga` imprime no console a string “Acertou!”. Caso contrário, imprime “Errou!”.

```
randnum:
      addi sp, sp, -4
      sw   ra, 0(sp)
```

```
      mv a1, a0
      li a7, 42
      ecall
      lw   ra, 0(sp)
      addi sp, sp, 4
      ret

      .data
msg_inicio:  .string  "Entre com um numero para adivinhar o valor: "
msg_acertou: .string  "Acertou!\n"
msg_errou    .string  "Errou!\n"

main:
     la a0, msg_inicio
      li a7, 4
      ecall
      li a7, 5
      ecall
      call joga
      li a7, 10
      ecall

joga:
      addi sp, sp, -4
      sw   ra, 0(sp)
      mv t0, a0
      li a0, 10
      call randnum
      beq  a0, t0, acertou
errou:
      la a0, msg_errou
      li a7, 4
      ecall
      j fim
acertou:
      la a0, msg_acertou
      li a7, 4
      ecall
fim:
      lw   ra, 0(sp)
      addi sp, sp, 4
      ret
```

7. Escreva uma função chamada `swap` utilizando o Assembly do RISC-V. A função recebe em a0 o endereço inicial de um vetor de inteiros, em a1 um índice do vetor e em a2 outro índice do vetor. A função deve trocar de posição os valores presentes em cada um dos índices informados para a função no vetor.

```
swap_vector:
  slli a1, a1, 2
  slli a2, a2, 2
```

```
add  t0, a0, a1
lw   a3, 0(t0)
add  t1, a0, a2
lw   a4, 0(t1)
sw   a3, 0(t1)
sw   a4, 0(t0)
ret
```

8. Escreva uma função chamada `ordena` utilizando o assembly do RISC-V. A função recebe em a0 o endereço inicial de um vetor de inteiros, em a1 o tamanho do vetor. A função deve ordenar o vetor em ordem crescente de valores. A função `ordena` deve obrigatoriamente utilizar a função swap desenvolvida no exercício anterior.

```
menor_vetor:
  addi  t0, zero, 1        # inicializa contador
  lw    t1, 0 (a0)         # le posicao 0 do vetor (menor)
  add   t2, zero, zero     # menor indice = 0
  addi  t3, a0, 4
laco_teste:
  beq   t0, a1, fim_busca
  lw    t4, 0(t3)          # le posicao i
  ble   t1, t4, segue
  add   t2, zero, t0       # novo indice menor
  add   t1, zero, t4       # novo menor valor
segue:
  addi  t0, t0, 1          # atualiza contador
  addi  t3, t3, 4          # atualiza ponteiro para memoria
  j     laco_teste
fim_busca:
  add a0, zero, t1
  add a1, zero, t2
  ret

ordena:
  add   s6, zero, ra
  add   s0, zero, a0
  add   s1, zero, a1
  add   s4, zero, zero     # menor indice = 0
segue_ordena:
  beq   s4, s1, fim_ordena
  jal   menor_vetor
  beq   s4, a1, atualiza_var
        add   a2, zero, zero
  slli  s5, s4, 2
  add   a0, s0, s5
  jal   swap_vector
atualiza_var:
  addi s4, s4, 1
  slli s5, s4, 2
  add  a0, s0, s5
  sub  a1, s1, s4
  j    segue_ordena
```

```
fim_ordena:
  add    ra, zero, s6
  ret
```

9. Escreva uma função utilizando o Assembly do RISC-V que remove todas as vogais presentes em uma string. A função recebe em a0 o endereço inicial da string e ao final da sua execução devolve em a0 o endereço inicial da string sem as vogais. Dicas: a) utilize a chamada de sistema 9 para fazer a alocação de memória para nova string sem vogais; b) utilize as instruções lb (load byte) e sb (store byte) para manipular a string na memória; c) o caracter '\0' marca o final da string.

```
    .data
string_original:   .asciz "Exemplo de string com VOGAIS\n"
vogais:            .asciz "aeiouAEIOU"

    .text
    .globl main

main:

    # Imprimir a string original
    la a0, string_original   # a0 = endereço da string original
    li a7, 4                 # syscall 4 = print_string
    ecall

    call remove_vogais       # chama função para remover vogais
                             # nova string está em a0
    # Imprimir a nova string
    li a7, 4                 # syscall 4 = print_string
    ecall

    # Finalizar programa
    li a7, 10                # syscall 10 = exit
    ecall

# ---------------------------------
# Função: remove_vogais
# Entrada: a0 = endereço da string original
# Saída:  a0 = endereço da nova string sem vogais
# ---------------------------------
remove_vogais:
    mv t0, a0            # t0 = ptr original
    mv t1, a0            # t1 = ptr para contar o tamanho
    li t2, 0             # t2 = tamanho

conta_tamanho:
    lb t3, 0(t1)
    beq t3, zero, aloca
    addi t2, t2, 1
    addi t1, t1, 1
    j conta_tamanho
```

```
aloca:
    addi a0, t2, 1      # a0 = tamanho + 1
    li a7, 9            # syscall 9 = sbrk
    ecall
    mv t4, a0           # t4 = nova string
    mv t1, t0           # t1 = ptr original
    mv t5, t4           # t5 = ptr nova string

loop_filtra:
    lb t6, 0(t1)
    beq t6, zero, fim

    la s0, vogais       # t7 = lista de vogais

verifica_vogal:
    lb s1, 0(s0)
    beq s1, zero, copia
    beq t6, s1, pula
    addi s0, s0, 1
    j verifica_vogal

copia:
    sb t6, 0(t5)
    addi t5, t5, 1

pula:
    addi t1, t1, 1
    j loop_filtra

fim:
    li t6, 0
    sb t6, 0(t5)        # termina nova string com '\0'
    mv a0, t4           # retorna ptr nova string
    ret
```

10. O ISA do RISC-V possui vários formatos de instrução (tipo R, I, B, S, J, etc). Para cada uma das instruções codificadas em hexadecimal mostradas abaixo, apresente a instrução em assembly e classifique a instrução quanto ao formato (R, I, B, etc), classe de instrução (aritmética, lógica, desvio condicional, etc) e o modo de endereçamento (registrador, imediato, relativo ao PC, etc).

Código	Instrução	Formato	Classe	Modo de endereçamento
0x00C50513	addi x10 x10 12	I	Aritméticas	Imediato
0x10000517	auipc x10 0x10000	U	Aritméticas	Imediato
0x00A004B3	add x9 x0 x10	R	Aritméticas	Registrador
0x0122A023	sw x18 0 x5	S	Acesso Memória	Base + Deslocamento

0x400904B3	sub x9 x18 x0	R	Aritméticas	Registrador
0x0240006F	jal x0 36	J	Desvio Incondicional	Relativo ao PC
0x0005a503	lw x10 0 x11	I	Acesso Memória	Base + Deslocamento

11. Implementar um programa que lê dois valores inteiros. Após ler os valores o programa deve chamar uma função chamada `multiplica_1` que, realiza a multiplicação dos valores usando somas sucessivas e retorna o resultado. Após o retorno da função o programa deve imprimir resultado no programa principal. Após isso o programa deve chamar a função `multiplica_2`, que realiza a multiplicação usando deslocamento e soma.

    Pseudo-código:

```
Programa:
main{
  ler valor1
  ler valor2

  aux = multiplica_1 (valor1, valor2)
  imprime aux
  aux = multiplica_2 (valor1, valor2)
  imprime aux
}

#multiplica usando somas sucessivas
int multiplica_1 (int num_a, int num_b){
   int x, result = 0;
   for (x =0; x < num_b; x++)
       result = result + num_a;
   return result;
}

# multiplica usando soma e deslocamento
int multiplica_2 (int num_a, int num_b){
   int result = 0;
   while(num_b > 0){
     if((num_b & 1) == 1)
       result = result + num_a;

       num_a = num_a << 1;
       num_b = num_b >> 1;
   }
   return result;
}
```

```
      .data
msg_valorA:    .string    "Entre com o valor de A: "
msg_valorB:    .string    "Entre com o valor de B: "
msg_result:    .string    "\nResultado da Multiplição: "

      .text
main:
      la    a0, msg_valorA
      li    a7, 4
      ecall
```

```
      li   a7, 5
      ecall
      mv   t0, a0

      la   a0, msg_valorB
      li   a7, 4
      ecall

      li   a7, 5
      ecall
      mv   t1, a0

      addi  sp, sp, -8    #empilha argumentos
      sw    t0, 0(sp)
      sw    t1, 4(sp)

      call multiplica_1

      mv    t2, a0
      la    a0, msg_result
      li    a7, 4
      ecall

      mv    a0, t2
      li    a7, 1
      ecall

      call  multiplica_2

      mv    t2, a0
      la    a0, msg_result
      li    a7, 4
      ecall

      mv    a0, t2
      li    a7, 1
      ecall

      addi  sp, sp, 8 #desempilha argumentos

      li   a7, 10
      ecall


##############################################
## multiplicação método somas sucessivas
multiplica_1:
      lw    a0, 0(sp)
      lw    a1, 4(sp)

      li    a2, 0                   # zera flag
```

```
      bge    a1, zero, segue_mult1    # testa se B é negativo
      not    a1, a1                   #faz B ficar positivo
      addi   a1, a1, 1
      li     a2, 1                    #flag indicando que B era negativo

segue_mult1:
      li     s0, 0                      # s0 = resultado = 0
laco1:
      beq    a1, zero, fim_mult1
      add    s0, s0, a0
      addi   a1, a1, -1
      j      laco1
fim_mult1:
      beq    a2, zero, retorna_mult1     #Se B era negativo
      not    s0, s0                      # inverte sinal do Resultado
ficar negativo
      addi   s0, s0, 1
retorna_mult1:
      mv     a0, s0     # retorna resultado em a0
      ret

##########################################
## multiplicação método soma e deslocamento
multiplica_2:
      lw    a0, 0(sp)
      lw    a1, 4(sp)

    li     t0, 0                 # t0 = resultado = 0

laco2:
    beq    a1, zero, fim_mult2   # while(num_B > 0)

    andi   t1, a1, 1             # t1 = num_B & 1
    beq    t1, zero, pula_soma

    add    t0, t0, a0            # result += num_A

pula_soma:
    slli   a0, a0, 1             # num_A <<= 1
    srli   a1, a1, 1             # num_B >>= 1
    j      laco2

fim_mult2:
    mv     a0, t0                # retorna resultado em a0
    ret
```