DCC006: Organização de computadores I

Trabalho Prático #1

Professor: Daniel Fernandes Macedo e Omar Paranaiba Vilela Neto

Antes de começar seu trabalho, leia todas as instruções abaixo.

- O trabalho deve ser feito individualmente. Cópias de trabalho acarretarão em devida penalização às partes envolvidas.
- Entregas após o prazo serão aceitas, porém haverá uma penalização. Quanto maior o atraso maior a penalização.
- Submeta apenas um arquivo .zip contendo as suas soluções e um arquivo .txt com seu nome e matrícula. Nomeie os arquivos de acordo com a numeração do problema a que se refere. Por exemplo, o arquivo contendo a solução para o problema 1 deve ser nomeado 1.s. Se for solicitado mais de uma implementação para o mesmo problema, nomeie 1a.s, 1b.s e assim por diante.
- O objetivo do trabalho é praticar as suas habilidades na linguagem assembly. Para isso, você utilizará o *Venus Simulator* (https://www.kvakil.me/venus/). O Venus é um simulador de ciclo único que te permite enxergar o valor armazenado em cada registrador e seguir a execução do seu código linha a linha. O simulador foi desenvolvido por Morten Petersen e possui a ISA do RISC-V, embora apresente algumas alterações. Você pode utilizar o seguinte link: https://github.com/mortbopet/Ripes/blob/master/docs/introduction.md para verificar as modificações da sintaxe ISA utilizada pelo simulador. Note que no livro e material da disciplina os registradores são de 64 bits, mas o simulador utiliza registradores de apenas 32 bits. Para utilizar o simulador basta você digitar seu código aba *Editor* e para executá-lo basta utilizar a aba *Simulator.*
- A correção do trabalho prático usará o simulador, e será feita de forma automatizada. Portanto, é crucial que vocês **empreguem as convenções de chamada de procedimento definidas no simulador**. Isso irá permitir que o trabalho desenvolvido seja avaliado corretamente (por exemplo, irei usar registradores “sx”, que são salvos pelo chamador, para realizar a contabilização automática de testes que foram executados corretamente. em outras palavras, caso seu procedimento use registradores "sx" eles deverão ser salvos na pilha). Para cada uma das questões, iremos definir um arquivo de base, onde está marcado a partir de qual ponto vocês deverão fazer o seu código. A avaliação irá considerar somente o que estiver escrito dentro daqueles limites (pois no momento da correção iremos alterar o início e fim do código fonte para fazer a correção).
- Eventuais testes apresentados nesta documentação são somente para indicar a funcionalidade a ser desenvolvida. O código dos alunos deve funcionar corretamente para todo e qualquer caso, inclusive aqueles que não estão previstos nos códigos, desde que sigam as especificações do trabalho. Façam testes além dos que estão descritos nesta documentação.

Problema 1: Números pares e ímpares (prob1.s) (5 pontos)

Escreva um procedimento que conte a quantidade de números pares e ímpares em um vetor. O seu procedimento deverá ter o nome “contador”, e irá receber os seguintes parâmetros.

- t2: endereço do vetor
- t3: tamanho do vetor

Você deverá salvar o resultado nos dois registradores de valores de retorno definidos no RISC-V (x10 para pares e x11 para ímpares).

Assuma que a memória onde os números do vetor serão escritos possui espaço suficiente para que todos eles sejam escritos.

Utilize o esqueleto a seguir para o seu arquivo **prob1.s (repare que a parte acima e abaixo do MODIFIQUE AQUI poderá ser alterada pelo professor/monitor no momento da correção:**

```
.data
vetor: .word 0 0 0 0

##### START MODIFIQUE AQUI START #####
#
# Este espaço é para você definir as suas constantes e vetores auxiliares.
#
##### END MODIFIQUE AQUI END #####

.text
jal x1, contador
addi x14, x0, 2 # utilizado para correção
beq x14, x10, FIM # Verifica # de pares
beq x14, x11, FIM # Verifica # de ímpares

##### START MODIFIQUE AQUI START #####
contador: jalr x0, 0(x1)

##### END MODIFIQUE AQUI END #####

FIM: addi x0, x0, 1
```

Problema 2: Ajuste salarial (prob2.s) (5 pontos)

Este programa irá exercitar um conceito muito importante no assembly: a gestão da pilha de chamadas de subrotinas de forma correta, que é necessária para desenvolvermos programas em que um procedimento chama outro procedimento ou para que seja possível usar bibliotecas de terceiros. Em outras palavras, você terá que manipular o espaço da memória denominado *stack* (pilha de funções) para armazenar corretamente o endereço de retorno para as próximas instruções.

O seu papel é desenvolver um programa que realiza reajuste de 50% nos salários de funcionários de uma determinada empresa. Além disso, você deve contabilizar a quantidade de salários que ficaram acima de um determinado valor. Para isso você deve implementar dois procedimentos.

- main: função principal a qual internamente chama a subrotina "aplica_reajuste".
- aplica_reajuste: responsável por aplicar reajuste de 50% nos salários.

Ao final, o procedimento main deve retornar a quantidade de salários que ficaram acima de um determinado valor.

O procedimento main deve receber os seguintes argumentos:

- a0: endereço do vetor
- a1: tamanho do vetor
- a2: limiar desejado

Você deve armazenar a quantidade de salários acima do limiar no registrador **t0**, para posteriormente ser comparado/verificado.

```
   .data
         vetor:  .word 200, 190, 340, 100 # exemplo

   ##### START MODIFIQUE AQUI START #####
   #
   # Este espaço é para você definir as suas constantes e vetores auxiliares.
   #
   ##### END MODIFIQUE AQUI END #####

   .text
       jal ra, main
# Ao final do reajuste (aplica_reajuste) você deve retornar o programa para a
próxima instrução abaixo, que consiste na correção/verificação.

    ##### START INSTRUÇÃO DE CORREÇÃO/VERIFICAÇÃO #####
    # utilizado para correção (considerando um limiar de 200 para o vetor de
exemplo após a aplicação do reajuste.

   addi a4, x0, 3 # configurando a quantidade de salários acima do limiar de 200.

    beq a4, t0, FIM # Verifica a quantidade de salários acima do limiar.

    ##### END INSTRUÇÃO DE CORREÇÃO/VERIFICAÇÃO #####

       main:
   ##### START MODIFIQUE AQUI START #####

           jal ra, aplica_reajuste

   ##### END MODIFIQUE AQUI END #####
   # dica, você deve salvar o endereço da primeira chamada (em stack (sp)) para
   depois recuperá-lo.

           aplica_reajuste:
   ##### START MODIFIQUE AQUI START #####


   ##### END MODIFIQUE AQUI END #####

   FIM: addi x0, x0, 1
```

Dicas e sugestões

- Não deixe o trabalho para o último dia. Não viva perigosamente!
- Comente seu código sempre que possível. Isso será visto com bons olhos.