Projeto 2: RISC-V Assembly

Caso queira verificar o arquivo original:
https://inst.eecs.berkeley.edu/~cs61c/fa18/labs/3/

Objetivos

Neste projeto você irá praticar a executar e debugar um código assembly para RISC-V. Você também irá escrever funções em RISC-V chamando as funções de forma correta e ter uma ideia de como traduzir o código em C para RISC-V.

Obtendo os arquivos

Segue o link dos arquivos para o projeto:
https://drive.google.com/file/d/1uM8SS04nuAuCLxSceaaxWS7zi7e1kISQ/view?usp=sharing

Introdução à Assembly com o simulador RISC-V

Neste projeto você aprenderá um pouco da linguagem assembly para RISC-V, que é uma linguagem de baixo nível muito próxima da linguagem de máquina. Para contextualizar, o GCC (compilador) primeiro reescreve nosso código para assembly (por exemplo, x86, ARM) e somente a partir disso que o GCC transforma o código em código de máquina (binário).

Nós iremos trabalhar com arquivos de programas assembly de RISC-V, cada um que tem o formato de extensão ".s". Para executá-los, precisaremos usar o Venus um simulador de RISC-V que você pode encontrar no link [https://thaumicmekanism.github.io/venus/]. Também existe uma versão de Venus no formato ".jar" que você pode encontrar no link [link quebrado, falar com o professor].

O básico sobre Assembly/Venus:

- Insira seu código na aba "Editor";
- Os programas iniciam da primeira linha independentemente do label. Isso significa que a função main deve ser colocada primeiro;
    - Nota: Às vezes nós queremos alocar previamente alguns itens na memória antes do programa começar a ser executado (veja mais na Parte 1 abaixo). Como essa alocação não é um código real, podemos colocá-la antes da função principal.
- Os programas são finalizados com o comando "ecall" com argumento de valor 10. Este é o sinal para o programa parar. As instruções "ecall" são análogas à "Chamadas de Sistema" e nos permitem fazer coisas como printar no console ou solicitar pedaços de memória heap (organização de memória mais flexível que permite o uso de qualquer área lógica disponível);
- Labels terminam com dois pontos ":";
- Comentários começam com uma hashtag "#";
- Você não pode colocar mais que uma instrução numa única linha;
- Quando você terminar de editar, clique na aba "Simulator" para preparar para execução.

Para os exercícios a seguir, por favor salve seu código completado em um arquivo no seu computador. Caso contrário, nós não teremos prova de que você completou o exercício.

Parte 1: Familiarizando você com Venus

Começando:

1. Cole o conteúdo do arquivo “ex1.s” no editor Venus;
2. Clique na aba “Simulator” e clique no botão “Assemble & Simulate from Editor”. Isso irá preparar o código que você escreveu para execução. Se você clicar de volta na aba “Editor”, sua simulação será resetada.
3. No simulador, para executar a próxima instrução, clique no botão “step”;
4. Para voltar uma instrução, clique no botão “prev”;
5. Para rodar o programa até seu fim, clique em “run”;
6. Para resetar o programa clique no botão “reset”;
7. O conteúdo de todos os 32 registradores estão na parte direita da tela e a saída do console está embaixo.
8. Para visualizar o conteúdo da memória clique em “Memória” na aba da direita. Você pode navegar para diferentes porções da memória usando o menu suspenso na parte de baixo.

Agora cole o conteúdo de “ex1.s” no Venus e registre suas respostas para as seguintes perguntas (para algumas delas será necessário que você execute o código RISC-V usando a aba “Simulator” do Venus).

1. O que os diretórios “.data”, “.word” e “.text” significam (qual a utilidade deles)?
   1.1. Dica: pense sobre diferentes sessões de memória.
2. Execute o programa até ser completo. Que número é printado no output? O que esse número representa?
3. Em que endereço “n” é armazenado na memória?
   3.1. Dica: olhe para o conteúdo dos registradores
4. Sem editar o código (sem navegar na aba “Editor”), faça o programa calcular o 13º número da sequência de Fibonacci (o primeiro elemento sendo 0) modificando manualmente o valor de um registrador. Você pode achar útil passar primeiro pelo código. Se você preferir olhar para os valores decimais, mude a opção “Display Settings” na parte de baixo.

Parte 2: Traduzindo de C para RISC-V

Abra os arquivos ex2.c e ex2.s. O código dado em assembly (arquivo “.s”) é a tradução do programa em C para RISC-V.

Agora, encontre/explique os seguintes componentes deste arquivo assembly:

- O registrador representando a variável “k”;
- O registrador representando a variável “sum”;
- Os registradores atuando como ponteiros para os arrays “source” e “dest”;
- O código em assembly para o loop encontrado no código em C.
- Como os ponteiros são manipulados no código em assembly.

Parte 3: Fatorial

Nesta parte, você vai implementar a função fatorial em RISC-V. Essa função recebe um único parâmetro inteiro `n` e retorna `n!`. Um esboço dessa função pode ser encontrado no arquivo `factorial.s`.

Você só precisa adicionar instruções abaixo do label fatorial, e o argumento que é passado para a função é configurado para ser localizado no label `n`. Você pode resolver esse problema usando tanto recursão quanto iteração.

Testando

Como uma checagem de sanidade, você deve ter certeza que sua função retorna corretamente `3! = 6`, `7! = 5040` e `8! = 40320`. Você tem a opção de testar isso usando a versão online do Venus, mas você pode usar a versão .jar pra testar localmente. Perceba que você deverá ter o java instalado para executar esse comando.

```
$ java -jar venus-jvm-latest.jar factorial.s
```

A versão .jar do Venus que você baixou aqui.

Parte 4: Chamada de função RISC-V com map

Essa parte usa o arquivo `list_map.s`.

Agora você vai completar uma implementação de `map` em listas encadeadas em RISC-V. Nossa função será simplificada para mudar a lista no local, ao invés de criar e retornar uma nova lista com os valores modificados.

Será de grande ajuda olhar o RISC-V green card para completar esse exercício. Se você encontrar alguma instrução ou pseudo-instrução que você não conhece, use isso como fonte.

Nosso processo `map` receberá dois parâmetros: o primeiro parâmetro será o endereço do nó inicial de uma lista simplesmente encadeada, cujos valores são inteiros de 32 bits. Então, em C, a estrutura será definida como:

```
struct node {
      int value;
      struct node *next;
};
```

Nosso segundo parâmetro será o **endereço de uma função** que recebe um int como argumento e retorna um int. Nós usaremos a instrução de RISC-V `jalr` para chamar essa função nos valores dos nós da lista.

Nossa função map irá recursivamente percorrer a lista, aplicando a função para cada valor da lista e guardando o valor retornado naquele nó correspondente. Em C, a função seria algo assim:

```
void map(struct node *head, int (*f)(int))
{
 if(!head) { return; }
 head->value = f(head->value);
 map(head->next,f);
}
```

Se você nunca viu o tipo de declaração `int (*f)(int)` antes, não se preocupe muito com isso. Basicamente isso significa que f é um ponteiro para uma função, a qual, em C, pode então ser usada exatamente como qualquer outra função.

Existem exatamente nove (9) marcadores (8 no map e 1 no main) no código disponibilizado onde diz "`YOUR CODE HERE`".

Agora complete a implementação do map preenchendo cada um desses marcadores com o código apropriado. Além disso, forneça um exemplo de chamada para map com square (quadrado do número) como o argumento da função. Existem comentários no código que explicam o que deve ser obtido em cada marcador. Quando você completar essas instruções, a execução do código deve fornecer a seguinte saída:

```
9 8 7 6 5 4 3 2 1 0
81 64 49 36 25 16 9 4 1 0
```

A primeira linha é a lista original, e a segunda linha é a lista modificada depois que a função map (nesse caso square) é aplicada.

Testando

Para testar isso localmente, execute o seguinte comando no terminal (parecido com aquele para o factorial.s):

```
$ java -jar venus-jvm-latest.jar list_map.s
```

Como submeter seu trabalho

Usando o Google Classroom, você irá submeter suas respostas para as Partes 1 e 2 em arquivos de texto chamados `part1.txt` e `part2.txt`. Para as Partes 3 e 4, você precisará submeter suas versões modificadas do `factorial.s` e `list_map.s`