Insper

Elementos de Sistemas - Projeto H - Ferramental - Assembler

Rafael Corsi - rafael.corsi@insper.edu.br

Abril, 2018

TODO de inicialização: Scrum Master:

1. Atualizar arquivo ScramMaster.txt
2. Atualizar repositório com o upstram
3. Adicionar o novo script de teste ao travis : F-Assembly/scripts/testeAssembly.py
4. Criar projeto no github
5. Atribuir tarefas e acompanhar o desenvolvimento

Esse projeto tem como objetivo introduzir a linguagem assembly e a arquitetura do computador Z01.

Projeto F - Assembler

Prerrequisitos

- Java JDK 9 >
- Maven
  - https://maven.apache.org/install.html
- IDE
  - Eclipse
  - Intellij (de graça via e-mail insper)

Entendendo a Organização do Projeto

A pasta do projeto H no repositório Z01, possui a seguinte estrutura:

1. scripts: Scripts em python que automatizam a execução dos testes;
   - compileNasmMyAssembler : Compila os nasms do projeto F-Assembly com o assembler do grupo e salva o resultado em bin/hack/

    - testeAssemblyMyAssembler : Compila os nasms com o assembler do grupo e executa a simulação no Z01 a fim de verificar os resultados.
    - genJAR : Gera um Jar que será utilizado pelos testes anteriores a partir das fontes em Assembler/src/main/ -> Salva em Assemler/Z01-Assembler.jar
2. bin/hack/ : Arquivos .hack convertidos via Z01-Assembler.jar
3. src/nasm/*.nasm: Arquivos ASSEMBLY que serão implementados pelo grupo;
4. tests/tst/*: Arquivos que realizam o teste nos arquivos códigos nasm.

Testes

É disponibilizado dois tipos de testes : **Unitário** para as classes em java e de **Integração** para o Assembler como um todo. Os testes unitários das classes estão localizados em **Assembler/src/tsts/** e pode ser executado de duas maneiras :

1. Via IDE (Eclipse / Intellij)
2. Via maven na geração do jar (**genJAR.py**)

Já o teste de integração que considera como as classes foram utilizadas para a geração do Assembler é executado via script **testeAssemblyMyAssembler.py**, executando os seguintes passos :

1. Gera o jar (genJAR.py)
    - input : Assembler/src/main/java/assembler/*.java*
    - output : Z01-Assembler.jar
2. Compila os nasms
    - input : F-Assembly/src/nasm/*.nasm*
    - output : bin/hack/*.mif/.hack*
3. Executa os testes no hardware (usando o hardware de referência)
    - input : tests/
    - output : tests/tst/ / *_end.mif*
4. Compara resultado com esperado
    - input : tests/tst/ / *_tst.mif*
    - output: Terminal

Carregando o projeto

Para carregar o projeto na IDE de sua preferência (Intellij ou Eclipse) basta importar um projeto do tipo maven.

Eclipse

- *File -> Import -> Existing Maven Project*

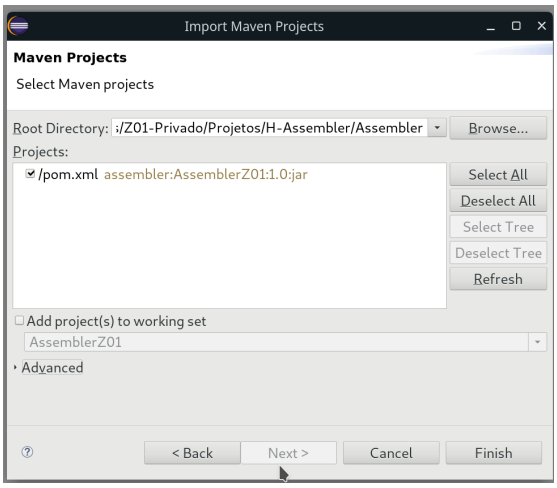

Figure 1: Importando

Problema com codificação dos acentos pelo Eclipse ? Try this

1. Window > Preferences > General > Content Types, set UTF-8 as the default encoding for all content types.
2. Window > Preferences > General > Workspace, set "Text file encoding" to "Other : UTF-8".

(ref: https://stackoverflow.com/questions/9180981/how-to-support-utf-8-encoding-in-eclipse)

Executando os testes unitários

Para executar os testes unitários no eclipse basta clicar em :

- *Run -> coverage*

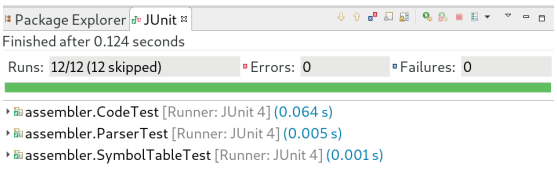

Figure 2: Testes

Nesse caso, nenhum teste foi executado porque os métodos não foram implementados.

Executando os testes de integração

Para executar os testes de integração : nasm -> Assembler -> hack -> Hardware (os mesmos usados no projeto F-Assembly), basta executar o script :

```
python testeAssemblyMyAssembler.py
```

Esse teste irá compilar os arquivos nasm do projeto F-Assembly com o assembler do projeto H-Assembler e irá executar os testes para verificar se o resultado é o esperado.

> **!** É necessário o maven instalado para executar.

Passo de execução:

1. gera .jar a partir dos códigos em java :
   - Assebler/Z01-Assembler.jar
2. monta .hack a partir dos .nasm localizados no projeto F
   - utiliza-se o .hack para criar o .mif
3. executa a simulação com o hardware Z01
4. verifica se o resultado é o esperado

Desenvolvimento guiado

> **!** Todos devem participar !

Essa etapa do ferramental serve para introduzir o desenvolvimento de uma dos métodos da classe **Code**, vamos trabalhar com o método :

```
/**
 * Retorna o código binário do mnemônico para realizar uma operação de jump (salto).
 * @param  mnemnonic vetor de mnemônicos "instrução" a ser analisada.
 * @return Opcode (String de 4 chars-bits) com código em linguagem de máquina para a ins
 */
public static String jump(String[] mnemnonic) throws InvalidJumpException {}
```

Essa classe recebe uma string referente ao comando de jump :

- “jmp”, “jge”, “jg”, ...

E deve retornar o binário correspondente aos bits j2, j1 e j0 do comando de jump :

- “111”, “011”, “010”, ...

> **!** Consulte o documento do instruction Set do Z01 para gerar esses comandos. (G-Computador/Instruction Set Z01.pdf)

Essa classe possui como entrada um vetor de Strings (mnemnonic) que é a instrução já parseada, por exemplo:

- {“movw”,“(%A)”,“%D”}
- {“jmp”}
- {“decw”,“%A”}

Note que o tamanho do array da string *mnemnomic* pode variar conforme o comando que será “analisado”.

O método deve retornar uma exceção quando receber como parâmetro um mnemonico incompatível com o definido pelo instruction set, por exemplo, se um mnemmonic for vazio {“”} o método deve lançar a exceção:

```
throw new InvalidJumpException();
```

Começando :

Vamos implementar algo bem simples, que está errado mas vai servir para entendermos o fluxo, modifique o código com o exemplo a seguir :

```
public static String jump(String[] mnemnonic) {
    String jumpBin = "";

    jumpBin = "111";

    return jumpBin;
}
```

Esse exemplo, não verifica qual a instrução de jmp que está chegando, apenas gera “111” nos bits : j2,j1,j0.

Debugando - teste unitário

Para debugar o projeto durante a fase de desenvolvimento basta inserir um *breakpoint* na linha que deseja testar, no nosso caso vamos inserir um na linha `String jumpBin = " "` :

> De duplo clique na barra azul na linha que deseja debugar, inserindo assim um break-point

Agora devemos executar o teste unitário em modo debug, forçando o mesmo a parar a execução quando chegar no breakpoint :

Agora podemos analisar o método e suas variáveis :

Controlando a execução via os comandos na barra superior:

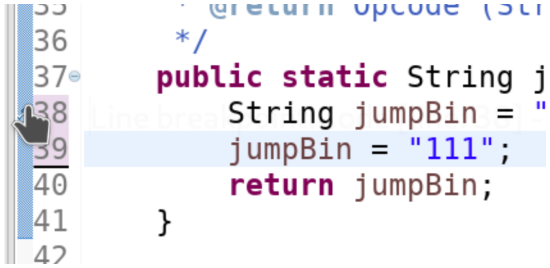

Figure 3: Eclipse Break Point

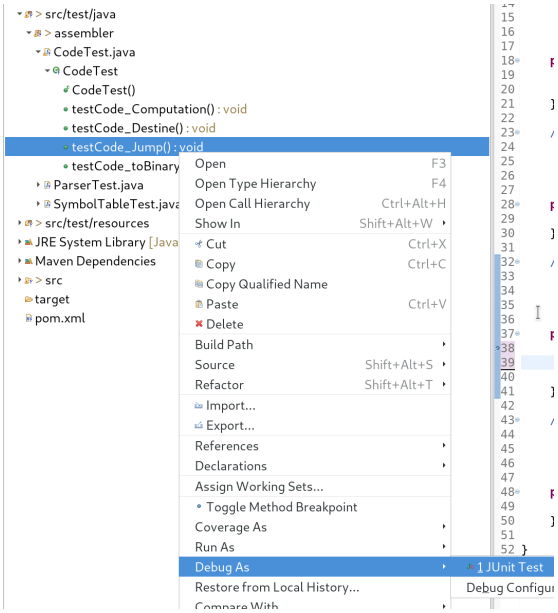

Figure 4: Eclipse run test debug

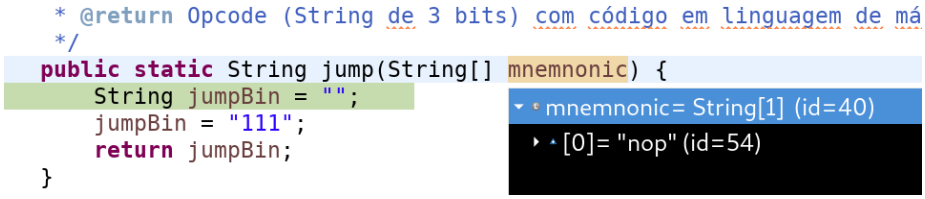

Figure 5: Estado atual

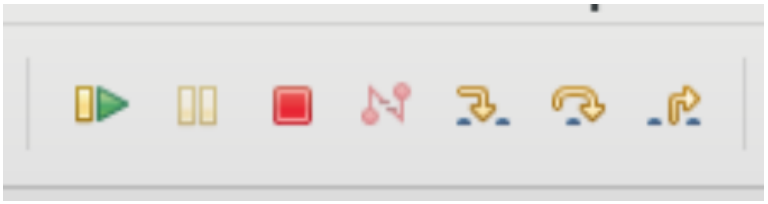

Figure 6: Comandos

> **!** Termine agora a implementação desse método, verificando e criando os comandos de jump com base no mnemmonic de entrada.