

Elementos de Sistemas - Projeto F - Ferramental - Assembly

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

Março, 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

Entendendo a Organização do Projeto

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

```
/G-Computador
  /script
    compileNasmMyAssembler.py
    testeAssemblyMyAssembler.py
    genJAR.py
  /Assembler
    /src/
      main/
      test/
    /bin/hack/
  /tests
    /tst/
      abs.
      add.
      ....
```

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 Assembler/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*

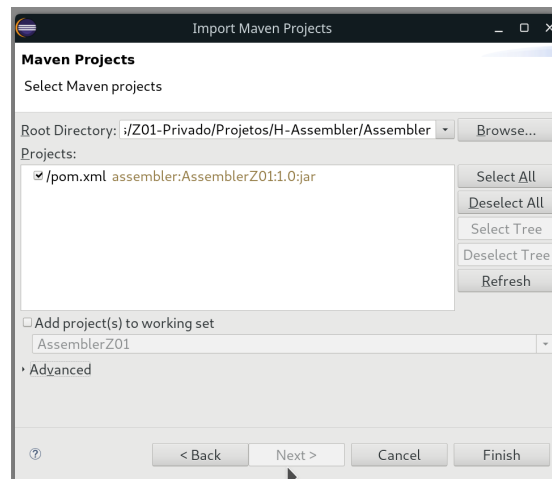


Figure 1: Importando

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*

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

Desenvolvimento guiado

! Todos devem participar !

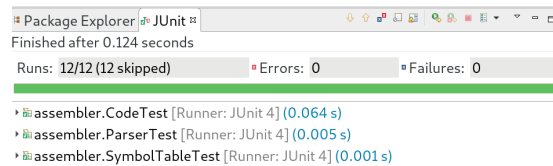


Figure 2: Testes

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 mnemonic vetor de mnemônicos "instrução" a ser analisada.
 * @return Opcode (String de 4 chars-bits) com código em linguagem de máquina para a in
 */
public static String jump(String[] mnemonic) 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 (mnemonic) que é a instrução já parseada, por exemplo:

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

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

O método deve retornar uma exceção quando receber como parâmetro um mnemonic incompatível com o definido pelo instruction set, por exemplo, se um mnemonic 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[] mnemonic) {  
    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 = " "` :

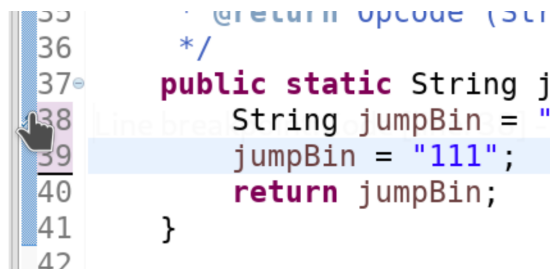


Figure 3: Eclipse Break Point

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:

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

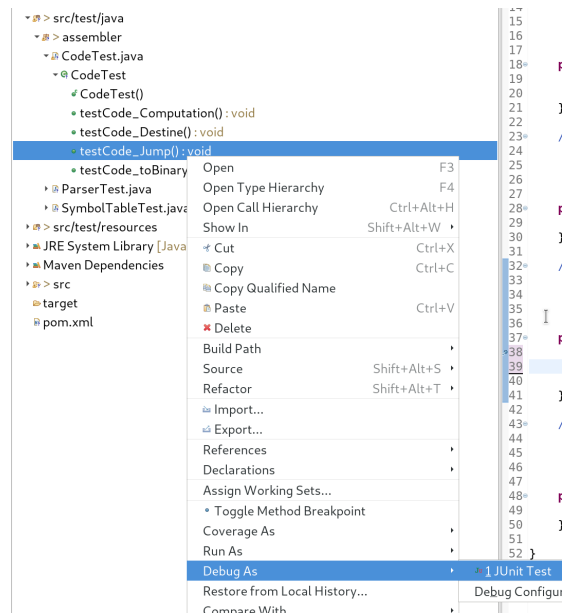


Figure 4: Eclipse run test debug

```

* @return Opcode (String de 3 bits) com código em linguagem de má
*/
public static String jump(String[] mnemonic) {
    String jumpBin = "";
    jumpBin = "111";
    return jumpBin;
}

```

▼ *mnemonic= String[1] (id=40)
 ▶ [0]= "nop" (id=54)

Figure 5: Estado atual

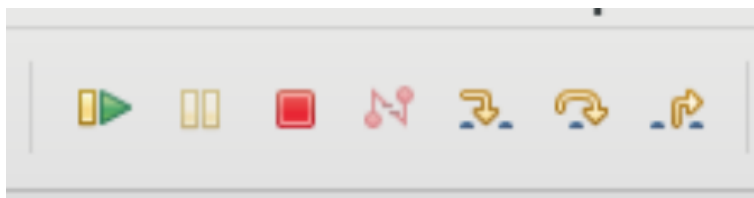


Figure 6: Comandos