Pseudo Assembler Overview

- A Pseudo Assembler program contains .data and .code sections.
- All variables should be declared at the top of the program in the .data section.
- All other code should be in the .code section.
- Registers names should be prefixed with either ri or rs. Registers ri1, ri2, … are int registers. Registers rs1, rs2, … are string registers. Should not use too many registers in total (<= 16).

A program contains data and code sections. All variable declarations go in the code section and all other code goes in the code section.

Pseudo Assembler Instructions

Instruction Format	**Example Call**	**Description**
;	; My comment	Comment. Any characters that appear after a ; on a line are ignored.
.data	.data	Variable declaration section. This should be the first section. All variables should be declared in this section.
.code	.code	Code section. All program code except for variable declarations goes in this section. It should appear after the .data section.
var type id	var int x	Declares an int variable. The variable x is declared in the example.
	var string y	Declares a string variable. The variable y is declared in the example.
loadintvar reg, var	loadintvar ri1, x	Loads an int register from an integer variable. Register ri1 is loaded with the value from variable x.
loadstringvar reg, var	loadstringvar rs1, x	Loads a register from a string variable. Register rs1 is loaded with the value from variable x.
loadintliteral reg, int	loadintliteral ri1, 77	Loads a register with an integer literal value. Register ri1 is loaded with the value 77.
loadstringliteral reg, string	loadstringliteral rs1, "abc"	Loads a register with a string literal value. Register rs1 is loaded with the value "abc".
storeintvar reg, var	storeintvar ri1, x	Stores an integer from a register into an integer variable. The value in register ri1 is stored in the variable x.
storestringvar reg, var	storestringvar rs1, x	Stores a string from a register into a string variable. The value in register rs1 is stored in the variable x.
inc reg	inc ri1	Increment register value. Only works on int registers. The value in register ri1 is incremented by 1.
dec reg	dec ri1	Decrement register value. Only works on int registers. The value in register ri1 is decremented by 1.
add reg, reg, reg	add ri1, ri2, ri3	Adds register values. Only works on int registers. The value in register ri1 is added to the value in register ri2 and the result is stored in register ri3.

mult reg, reg, reg	mult ri1, ri2, ri3	Multiples register values. Only works on int registers. The value in register ri1 is multiplied with the value in register ri2 and the result is stored in register ri3.
printi var or reg	print x print ri1	Prints an int value from a variable or from a register. The value given to print is displayed in the console window.
prints var or reg or stringliteral	print x print rs1 print "Hello"	Prints a string value from a variable or from a register or from a string literal. The value given to print is displayed in the console window.
:id	:label1	Label instruction. Used as the destination for jumping instructions (bne, be, bgt, blt).
branch id	branch label1	Branch. Unconditionally jumps to the given label id.
bne reg, reg, id	bne ri1, ri2, label1	Branch Not Equal. Only works on int registers. Jumps to the given label id if the values in two registers are not equal. If the value in ri1 is not equal to the value in ri2 it jumps to label1.
be reg, reg, id	be ri1, ri2, label1	Branch Equal. Only works on int registers. Jumps to the given label id if the values in two registers are equal. If the value in ri1 is equal to the value in ri2 it jumps to label1.
bgt reg, reg, id	bgt ri1, ri2, label1	Branch Greater Than. Only works on int registers. Jumps to the given label id if the value in the first register (ri1 in this example) is greater than the second register (ri2 in this example). So, if the value in ri1 is greater than the value in ri2 it jumps to label1.
blt reg, reg, id	blt ri1, ri2, label1	Branch Less Than. Only works on int registers. Jumps to the given label id if the value in the first register (ri1 in this example) is less than the second register (ri2 in this example). So, if the value in ri1 is less than the value in ri2 it jumps to label1.

Pseudo Assembler Sample Program

The following program uses both int and string registers and variables and has comments.

```
.data
var int x
var string y

.code

; Put an int in ri1 and store in variable x
loadintliteral ri1, 88
storeintvar ri1, x

; Print an int from a variable and a register
printi x
printi ri1

; Put a string in rs1 and store in variable y
loadstringliteral rs1, "Hello world"
storestringvar rs1, y

; Print a string from a variable and a register
prints rs1
prints y
```

Parsing and Running Pseudo Assembler Program

Do the following to compile and run Pseudo Assembler program.

- Make sure you install the Pseudo Assembler library in your local Maven repository and add its Maven dependency to your IntelliJ project. Here is the Maven dependency:
  ```
  <dependency>
    <groupId>org.example</groupId>
    <artifactId>PseudoAssemblyObf</artifactId>
    <version>1.0</version>
  </dependency>
  ```
- Create an instance of PseudoAssemblyWithStringProgram. The constructor takes the following parameters:
  - The Pseudo Assember code to compile and run.
  - The output class name. It creates a Java bytecode class with this name behind the scenes. This class will include a main method. This main method will have the bytecode for the Pseudo Assembler program inside of it. A .class file with the output class name will be created using this name. It will automatically append ".class" to the output class name. The .class file will be created when PseudoAssemblyWithStringProgram .generateBytecode is called (see below).
  - The package name for the output class. This is necessary because it needs a fully qualified class name when creating a class. It will automatically add this package name as a prefix to the output class name you specify.
  - Class root directory. The root directory for where the .class file will be created. The sample code below shows how to set this up. The sample code will create the .class file in your project's target/classes directory. The package name you specify will be a subdirectory of this class root directory. This subdirectory is where the .class file will be located.
  - Number of virtual int registers. You should keep this value <= 8.
  - Number of virtual string registers. You should keep this value <= 8.
- Call the parse method on the PseudoAssemblyWithStringProgram instance. This will return true or false depending on if the parse was successful.
- If the parse was successful, call generateBytecode on the PseudoAssemblyWithStringProgram instance. This will generate a Java bytecode .class file. The name of the file will be the output you specified in the call to the PseudoAssemblyWithStringProgram constructor appended with ".class".
- If the parse was successful, call run on the PseudoAssemblyWithStringProgram instance. This will execute the Java bytecode .class file. The run method takes a PrintStream as a parameter. The program output from running the .class file will be written to the PrintStream. Note: You can create a PrintStream that is connected to System.out to write data to the console.
- Call getAllParseMessages on the PseudoAssemblyWithStringProgram instance to get the output messages from the call to parse. This will return a string. You can display the string to the user to help with debugging.

- Call getProgramListingWithLineNumbers on the PseudoAssemblyWithStringProgram to get the program listing. This will return a string. You can display the string to the user to help with debugging.

Sample Code to Create and Run a Pseudo Assembler Program

Below is Java code that will compile and run a Pseudo Assember program. The Pseudo Assembler code to compile and run is stored in a string. This string is passed as one of the parameter to the PseudoAssemblyWithStringProgram constructor.

```
String code = "";

code += ".data\n";
code += "var int x\n";
code += ".code\n";
code += "loadintliteral ri1, 88\n";
code += "storeintvar ri1, x\n";
code += "printi x\n";
code += "printi ri1\n";
```

Pseudo Assembler code is stored in a string. This code can be read in from a file instead of hardcoding it.

```
int numVirtualRegistersInt = 32;
int numVirtualRegistersString = 32;

String outputClassName = "MyProgram1";
String outputPackageNameDot = "mypackage";
String classRootDir = System.getProperty("user.dir") + "/" + "target/classes";
```

Variables which will be passed to PseudoAssemblyWithString Program constructor

```
PseudoAssemblyWithStringProgram pseudoAssemblyWithStringProgram = new
PseudoAssemblyWithStringProgram(
      code,
      outputClassName,
      outputPackageNameDot,
      classRootDir,
      numVirtualRegistersInt,
      numVirtualRegistersString
);

boolean parseSuccessful;
parseSuccessful = pseudoAssemblyWithStringProgram.parse();

if (parseSuccessful == true) {
   // Creates a Java bytecode class file
   pseudoAssemblyWithStringProgram.generateBytecode();

   // Run the Java bytecode class file and show output on the console
   PrintStream outstream = new PrintStream(System.out);
   pseudoAssemblyWithStringProgram.run(outstream);
}
```