

# **Introduction to Building Assemblers**

## **A Case Study**

By Carlo Bruscani and Victor Brusca

# **Forward**

Take a moment to review this section and make sure this book is right for you before you begin. This book is in a non-reflowable format. The structure of text is required in a lot of places so this was a necessity. The e-reader version of the text is small but should be more than readable. If it's an issue for you try the browser version of the book which has a larger format for larger screens. You'll want to have the book open on two screens or one screen for the book and one for the code.

## **Who is This Book For**

This book is for intermediate to advanced level software developers who have experience working with object oriented programming languages like C++, C#, Java, etc. The project that is reviewed in this text is entirely written in Java and the review is a case study of the implementation of the GenAsm assembler. Having some familiarity with Java syntax is beneficial but not necessarily required if you have a solid programming background.

# **Who is This Book NOT For**

This book is not for those who have introductory level programming experience or are not familiar with object oriented programming techniques. That being said if you are willing to put in the work and the time necessary to learn the code, including Java syntax, you will develop a strong understanding of Java and how it can be used to solve complex problems, like building an assembler. Also I should note that this book is not for someone who wants to learn assembly programming. Aside from reviewing all the opcodes in our target instruction set we won't be discussing assembly programming much at all.

# **What Software is Needed to Utilize This Book**

The software needed to utilize this book is all freely available and some of it is including in the project for convenience. The software list is as follows.

- NetBeans: The main IDE used to review code and sometimes JSON data files or assembly source code.  
<https://netbeans.apache.org/front/main/index.html>
- Notepad++: An awesome text editor used to view larger JSON data files and assembly source code.  
<https://notepad-plus-plus.org/>
- GameBoy Advance Emulator (Included): The emulator used to run the example assembly programs is mGBA. However, Visual Boy Advance is also included as a backup.  
<https://mgba.io/>  
<https://visualboyadvance.org/>

# **How to use This Book**

Because this book is a case study of an implemented program we spend a tremendous amount of time reviewing code. You won't be building an assembler step by step but rather reviewing an implementation of an

assembler, GenAsm, that support the ARM Thumb-1 instruction set. In order to do this and maintain your sanity you should have two screens to look at when reading this book. On one screen you should have either the IDE open with the Java code being reviewed visible or a copy of the book with the pertinent section visible as you read through the code review.

## Dedication

Firstly, this book is dedicated to all those who made mistakes in life. Learn what you can from it but don't let it define you. Pick yourself up, dust yourself off and keep going. Secondly, this book is dedicated to my little brother Victor.

# Chapter 1: Introduction

Welcome to Chapter 1 of the “Introduction to Building Assemblers” text. This book is designed as a detailed review of an instruction set, ARM Thumb-1, a JSON model of the instruction set, and an extensible assembler project in Java that allows you to define your own instruction set and lastly an ARM Thumb-1 assembler. The project supports a preprocessor, lexer, tokenizer, assembler, and linker. The GenAsm assembler generates a number of output files at different stages of the assembly process to support the educational aspects of the software.

As a life-long programmer I’ve always wanted to get closer to the metal and learn how to program in assembly. This text is the result of my exploration of writing an assembler from scratch. We’ll learn all about a specific instruction set, the ARM Thumb-1, and we’ll get to test our programs on hardware based on an ARM7TDMI processor. Specifically, we’ll use a Nintendo GameBoy Advance emulator to run the test programs we’ll explore in this text.

Furthermore, you’ll gain experience modeling the instruction set with JSON data files, and working with lexers, tokenizers, assemblers and linkers. The journey provided by this text will give you a deep insight into how software actually works. You’ll see first-hand how an assembler generates a

binary output file from source code by reviewing the Java implementation of the GenAsm assembler in detail.

What is more you'll be exposed to data modeling, file IO, project structure, class inheritance, encapsulation, and structure, as well as experience working with reflection and a data driven software model. The general approach to processing the assembly source code is as follows.

1. Assembly Source to Pre-processed Assembly Source (Preprocessor)
2. Pre-processed Assembly Source to Artifacts (Lexer)
3. Artifacts to Tokens (Tokenizer)
4. Tokens to OpCode/Directive Identified Tokens (Assembler)
5. OpCode/Directive Identified Tokens to Binary Output (Linker)

From the perspective of the data being processed by the assembler the following describes the general flow of data structures and their use.

1. Instruction set data is loaded from JSON data files.
2. Assembly source lines are loaded into token lines.
3. Token lines are compared to JSON data for an OpCode match.
4. Token lines are compared to JSON data for a Directive match.
5. Token lines and matching JSON data are used to generate binary representations of each line.
6. Binary representations are converted to bytes and written out.

In the next section we'll look at how to setup the development environment and compile the assembler project. But before we get into that I wanted to discuss a little bit about this text and how the code reviews are laid out. For the most part I will address every line of source code as part of a class-by-class review. These reviews consist of method reviews with the code listed in-line in the text.

Due to the difficulties maintaining absolute line numbers, any change to the text requires a significant rewrite of a portion of this text, even adding one line would cause some trouble. To this end we use local line numbering, meaning every method or block of code reviewed starts at line 1. You will often have to refer back to the code being described so it's imperative that

you setup an environment where you can reference the text and the source code easily. If you have two screens use one for source code and one for the book text.

If you don't have two screens, try to get two versions of the book in front of your eyes. For instance, a hard-copy and digital version, two e-reader windows opened, etc. This way you can quickly switch between code and text.

## Setting Up Your Environment

In this section we'll get your development environment up and running so that you can take the GenAsm project for a little test run and see it in action. This will give you an idea of what the project is all about. You'll also get some exposure to the assembler's source code and syntax, and generated output files. As an added bonus, because we're using the ARM7TDMI Thumb-1 instruction set, we'll be able to run our test programs on the GameBoy Advance emulator.

First off, we'll need to get an IDE installed and setup. This text uses the NetBeans IDE for Java project management. Navigate your favorite browser to the <https://netbeans.apache.org> website and download the latest long term support release from the website. Make sure you have the appropriate version of the Oracle JDK for the version of NetBeans you've just downloaded. You can download versions of the Oracle JDK from the <https://www.oracle.com/java/technologies/downloads/> website. You may need an account to download certain versions of the JDK.

This project by default uses JDK 11, which was a recent version at the time the software was being developed. The software doesn't use any advanced Java features so changing the project's target JDK should be quick and easy. If you need to update the target JDK the NetBeans IDE will prompt you to do so the first time you load up the project in the IDE.

You'll also need a good text editor. You can use the NetBeans IDE to edit your GenAsm assembler JSON data files, but I tend to prefer using Notepad++ for that purpose. I recommend navigating your favorite browser to

the Notepad++ site, <https://notepad-plus-plus.org>. Next, find and install the latest version of the software for your platform.

Once you have NetBeans, Java, and Notepad++ properly installed and configured, you'll need to download a copy of the project. You can do so at this URL, <https://github.com/vbrusca/GenAsm>. Download the project ZIP file and decompress it into the directory where you plan to keep your NetBeans projects. Once this step is complete open NetBeans and open the project you just created. You should see a project entry named 'GenAsm' in the list of open projects.

Next, we'll want to adjust the project's working directory. This helps when debugging the project through the IDE by maintaining the same working directory, and structure, you'd expect if you were executing it by hand. Right-click on the 'GenAsm' project entry in the 'Projects' panel and choose 'Properties.' A project properties popup will appear. Select the 'Run' entry from the list of categories.

Make sure the working directory for the 'GenAsm' project is set to the root directory of the project. On my computer I use the following directory, C:\Users\variable\Documents\GitHub\GenAsm. Set the 'Working Directory' property to the root of the 'GenAsm' project in your environment. Now that we have that worked out let's make sure the project compiles automatically when we save a change.

Select the 'Compiling' entry under the 'Build' category on the left-hand side of the project properties popup window. Once selected, notice there is a checkbox at the top of the window's main panel. Select the 'Compile on Save' checkbox at the top of the window and agree to any popup dialogs required to support the compile on save feature.

We're almost ready for our test run but there's one more path we'll need to edit before we can move on. Locate and open the GenAsm.java class file and direct your attention to the CFG\_DIR\_PATH static class field. This path is used as the root directory of the GenAsm project's 'cfg' folder. Change the path stored here so that it matches your DEV environment if need be. Make sure you end the path with a valid path separator character for your environment.

Before we move on to the next section let's make sure the project is working correctly. Right-click the 'GenAsm' project and select the 'Run' context menu entry. The project should run without error and output a bunch of text to standard output. You should see text in the output window for the default assembly program compiled by the GenAsm assembler.

#### ***Listing 1-1. Assembler Successful First Run***

```
LinkerThumb: RunLinker: Found 269 lines of linked assembly json objects
Utils: WriteObject: Name: Linked Assembly Source Code Lines
Found total bytes Big Endian: 542
Found total bytes Lil Endian: 542
```

A text listing showing the tail end of the assembler's output.

If everything has been configured correctly you should see some lines of output similar to those shown in the previous listing. If you are getting errors from your program run, take a moment to troubleshoot your project. If you're still having trouble uninstall and reinstall the NetBeans IDE and the 'GenAsm' project and carefully work through the configuration steps again. Otherwise, congratulations, you have everything setup correctly and you're ready to move on to the next section.

## **Our First Assembled Program**

Now that your development environment is all setup and configured correctly let's take a little crash course introduction to the GenAsm assembler. Navigate to the following file included in the GenAsm project and open it in Notepad++ or your favorite text editor.

`.\cfg\THUMB\TESTS\TEST_M_YourName\genasm_source.txt`

Take a moment to look over the file. The GenAsm syntax is a little bit unique but certainly looks similar to other assembly languages and their syntax. For unit testing and validation purposes I've included a version of every test program written in the assembly language syntax of the VASM assembler. There will be more on this topic to come. For now, it's enough to

know we can compare our test programs against a known, tried-and-true source.

Scroll down to line 164, where the `your_name` label is defined. Find and replace the text, `Your Name Here`, with your name. Try to keep the number of letters roughly the same as the original text. If you have a long name maybe an abbreviation is in order. Save the file. Now, open the NetBeans IDE, if need be, and open the `GenAsm` class. We're going to be editing the default configuration of the program so that it will compile the `TEST_M_YourName` program by default when provided no arguments.

Go to line 142 in the class where the `targetProgram` local variable is initialized. Change the value of the target program to, "TEST\_M\_YourName." You should see something similar to the following.

```
String targetProgram = "TEST_M_YourName";
```

Once this is done run the `GenAsm` program in the NetBeans IDE and you should see text similar to the following in the output window if the IDE.

#### ***Listing 1-2. Assembler Compile TEST\_M\_YourName Program***

```
LinkerThumb: RunLinker: Found 382 lines of linked assembly json objects
AssemblerThumb: WriteObject: Name: Linked Assembly Source Code Lines
Found total bytes BigEndian: 772
Found total bytes LittleEndian: 772
```

A text listing showing the tail end of the assembler's output.

I know that was a little anticlimactic but hang in there we'll get to the fun part soon. Next up, we're going to take a look at the assembler's output files. Keep in mind we're just doing a demonstration at this point. We'll cover this material in detail throughout the text. The `GenAsm` assembler was written for the purpose of demonstrating how to write an assembler. It is not an enterprise level piece of software by any means. As such, some aspects of the project will diverge from other assemblers. One such difference is the

number of output files generated by the GenAsm assembler. You'll see what I mean in the next section.

## Crash Course in Assembler Output

Now that we've assembled the first test assembly program. Let's take a look at the output generated by the assembler. In the NetBeans IDE, open the 'Files window', main menu -> windows -> files, and expand the 'cfg' folder. Next, expand the 'THUMB' folder, then the 'OUTPUT' sub-folder. This is the default directory where output for the example program is written. The output generated is separated for each test program by default. Open the 'TEST\_M\_YourName' folder in the project's 'OUTPUT' folder.

Note the number of files listed in the directory. The assembler generates a lot of output by default. This is really helpful when building your own instruction set for the assembler but has to be supported by your code. For the Thumb-1 implementation I made sure to keep output files at key stages in the assembly process so we can access them as we learn about building an assembler. The output files are as follows:

### *Listing 1-3. Assembler Output*

Preprocessor Output (Preprocessor):

1. **output\_pre\_processed\_assembly.txt**: Preprocessor output, preprocessor directives replaced.

Lexerizer, Tokenizer Output (Assembler):

1. **output\_lexed.json**: Lexerized output, text artifacts extracted.
2. **output\_tokened\_phase0\_tokenized.json**: Tokenized output, artifacts converted to tokens.
3. **output\_tokened\_phase1\_valid\_lines.json**: Basic line validation check.
4. **output\_tokened\_phase2\_refactored.json**: Collapse comment, group, and list tokens. Populate OpCode and Directive arguments. Determine data and code areas.
5. **output\_tokened\_phase3\_valid\_lines.json**: Strict line validation check.

6. **output\_tokened\_phase4\_bin\_output.json**: Convert tokens to binary representation in preparation for the linker.

Symbol Table Output (Assembler):

1. **output\_symbols.json**: A JSON representation of the symbols table loaded from the assembled source code.

Code, Data Area Output (Assembler):

1. **output\_area\_desc\_code.json**: A JSON representation of the code area defined in the assembly source code.
2. **output\_area\_desc\_data.json**: A JSON representation of the data area defined in the assembly source code, if any.
3. **output\_area\_lines\_code.json**: A JSON representation of the lines of assembly in the code area.
4. **output\_area\_lines\_data.json**: A JSON representation of the lines of assembly in the data area, if any.

Linked Lines, Binary Output (Linker):

1. **output\_assembly\_listing\_endian\_big.list**: A listing file for the big-endian binary output.
2. **output\_assembly\_listing\_endian\_big.bin**: A binary, big-endian, representation of the assembly source code.
3. **output\_assembly\_listing\_endian\_lil.list**: A listing file for the little-endian binary output.
4. **output\_assembly\_listing\_endian\_lil.bin**: A binary, little-endian, representation of the assembly source code.

A breakdown of the different outputs generated by the assembler and what step in the process generates them.

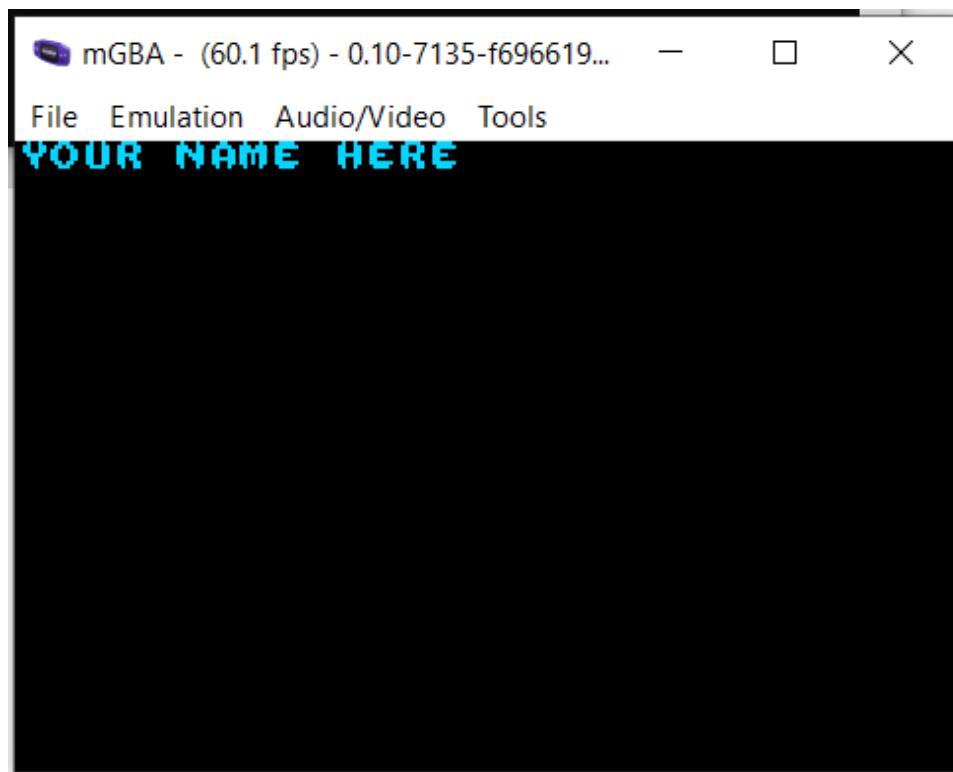
That's quite a bit of output! Let's look past most of the files and focus on the little-endian binary output,

'output\_assembly\_listing\_endian\_lil.bin' Because we're targeting the GameBoy Advance, ARM7TDMI based processor we're only going to use the little-endian binary as that is the default setup for the chosen hardware. In the next section we'll execute the binary output using a GameBoy Advance emulator.

# Executing the Generated Binary

In this section we're going to run the little-endian binary file you just created. We'll get to see our name on the GameBoy Advance emulator. From the 'TEST\_M\_YourName' output folder, locate and execute the 'run.bat' file. This will launch the mGBA emulator and run the program's binary output. If everything went to plan you should see the following on your screen.

*Listing 1-4. TEST\_M\_YourName in Action*



An image depicting the TEST\_M\_YourName program running in the mGBA emulator.  
Congrats on your first GenAsm assembled program.

Well, there you have it. We just edited, assembled, and ran our first GenAsm generated program and tested it on a GameBoy Advance emulator. Not too shabby. I hope this chapter piqued your interest in learning about building assemblers. Throughout this text I'll guide you through a detailed review of the code that powers the assembler so that you can gain insight and

understanding into the process and go on to build a great assembler of your own.

Through this process you'll gain a deeper understanding of what's going on under the hood when a program runs. You will literally be able to see the CPU instruction at each memory location and view the contents of the emulator's memory and registers. If you've spent most of your time coding with higher level languages this experience may be trans-formative for you. If you're used to lower-level programming languages or assembly languages themselves then you'll gain experience managing the data, structures, and processes necessary to compile assembly source code.

## Chapter Conclusion

That brings us to the conclusion of this chapter. We actually covered a fair amount of ground in this little introduction chapter. I hope you were impressed by the demonstration and eager to learn more about building an assembler. Let's take a look at what we've done so far:

1. **Setup your Environment:** We configured your development environment by installed NetBeans, the JDK, and Notepad++.
2. **Tested the Assembler:** We quickly tested the assembler by running and allowing the default program to assemble.
3. **Assembled our First Program:** We adjusted the GenAsm class to assemble the 'TEST\_M\_YourName' program.
4. **Reviewed GenAsm Output Files:** We did a quick, crash course, review of the assembler's output files and described the stage in the assembly process where the file is generated and what it contains.
5. **Executing the Generated Binary Output:** Last but not least we ran an assembled binary file on the GameBoy Advance emulator!

At this point in our journey, we're ready to get into the details of how the GenAsm assembler is built with a detailed review of the code that powers it.

But first, we need a little bit of background on the ARM Thumb-1 instruction set and a quick review of what opcodes are and how they are used to instruct the ARM CPU to do our bidding.

# **Chapter 2: Introduction to Assembly Programming**

In this chapter we're going to talk about assembly programming, the source code syntax of the GenAsm assembler, and the structure of the listing output file. I won't go into too much detail on the broader topic of assembly programming, but we'll make sure to cover the fundamental material. As mentioned above we'll also cover the syntax of the assembler's language, keywords, directives, etc.

We'll go over, in detail, the different types of tokens you can define and how you can define them. Lastly, we'll take a look at one of the assembler's key outputs, the listing file. I'll make sure to review the listing file's syntax including the line number, memory location, instruction code, and more. Well, we have a lot of work ahead of us so let's hop to it shall we.

## **Assembly Programming Fundamentals**

A computer program is, at its simplest, just a series of binary numbers. These numbers are fed, in series, to an MCU or CPU that is designed to work with and understands the binary numbers in the program. A specific action is then

taken by the controller unit, MCU or CPU. The amount of information that can be contained in one number, in the program's series of binary numbers, is dependent on the size of the number. For instance, an 8-bit MCU can only read numbers that are 8 bits and so the largest number that can be represented is 255.

Now, let's adjust our thinking a little. Let us consider these "binary numbers" that constitute a program to be instructions for the microprocessor. After all we've just stated that the numbers cause the microprocessor to take a specific, defined, action so it's not a leap to think of them as instructions. We give the MCU an instruction and it performs a specific action. I also loosely refer to an instruction as an opcode in this text.

Let's take a look at an example. In the following listing we can see a series of numbers. The MCU on the left is an 8-bit microprocessor and as such can only take one, 8-bit, instruction at a time. On the other hand, the MCU on the right is a 16-bit microprocessor, and as such it can take a two-byte, 16-bit, instruction at one time. Now it is important to note that even though in this example we're using the same series of bytes the instructions between the 8-bit and 16-bit versions of the program won't be the same. This was done to enforce the notion that a program is a series of binary numbers, the bit width of which is determined by the architecture of the microprocessor the program is designed to run on.

*Listing 2-1: Text Demonstrating MCU/CPU Reading Instructions*

<b>Program 1</b>	<b>Program 2</b>
00000000	0000000000000001
00000001	0000001000000100
00000010	MCU: 0000100000010000
00000100	
00001000	
MCU: 00010000	

A text representation of an 8 and 16-bit program. The "MCU" line entry indicates which instruction the processor is currently executing.

In the previously listed example, the MCU on the left, program 1, is an 8-bit MCU and it's processing a program that is constructed from 6, 8-bit, numbers. On the right-hand side of the listing is program 2, a 16-bit MCU. Notice that although the program consists of the same 6 bytes the MCU on the right processes 2-bytes, 16-bits, at a time. This means that the program on the left and the program on the right should not be thought of as equivalent even though they have the same binary encoding.

This is because program 1 is an 8-bit program and as such cannot use the same instructions, opcodes, as the 16-bit program. The takeaway here is that it's not enough to just have the binary program, you also need to understand the context of the program. What MCU/CPU was it designed to run on? Is it 8-bit, 16-bit, 64-bit? Is the microprocessor little or big endian? The answers to these questions will help you get a better understanding of the hardware and instructions you're working with. Notice we've sort of indicated that there will be a set of these "instruction" numbers that can be used to create a program. We'll refer to this set as an instruction set.

That brings us to our next topic. The instruction set! Each microprocessor comes with an instruction set from its manufacturer. The instruction set consists of the entire set of binary inputs that produce defined actions during an instruction's execution which occurs during a microprocessor cycle. Thus, based on what we've covered in this section a computer program for a specific MCU/CPU consists of a series of instructions, from a set of known instructions provided by the hardware manufacturer. Now that we have that in mind, we can sort of imagine writing a program by connecting a series of fixed width binary numbers together in one file. Let's explore this concept a bit further.

## Binary Assembly Programming

Our first approach to solving the problem of how we should program the MCU/CPU is to just write out the binary instructions by hand. Why not? For an 8-bit computer this would be very easy. For a 64-bit computer it would be a little bit tedious, but we could still do it. From now on let's simply refer to the MCU/CPU as the CPU, but keep in mind we mean any programmable micro processing unit. Let's try our hand at it and use the example programs from the previous listing. For our purposes let's pretend these binary numbers constitute some small but meaningful piece of assembly.

### *Listing 2-2: Fake Sample Programs with Errors*

#### **Program 1**

```
00000000  
00000001  
00000010  
00000100  
00001000  
00010000
```

#### **Program 2**

```
0000000000000001  
0000001000000100  
00010000010000
```

A text representation of an 8 and 16-bit program. The bold entry indicates an error in the binary instruction.

Now I'm not being negative when I say that we should plan to make at least one error when typing out our binary assembly program. In the previous listing the error bit has been marked for us in bold. Think for a moment what it would be like if you were looking at a real binary listing with no emboldened erroneous bit. It would be difficult to impossible to spot an error using this approach. The following issues present themselves with this method of assembly programming.

1. Difficult to read, trace, understand, debug.
2. The code is not intuitive or descriptive.
3. This approach is error prone.

All of that being said people did actually program computers this way until there were enough resources to build an assembler to do it for them. That leads us to our next approach to programming the CPU, we'll change the base to something more descriptive. That way we won't be dealing with the repetitive 0's and 1's that make the binary assembly programming approach so error prone and difficult.

## **Hexadecimal Assembly Programming**

Due to the simplicity of base 2 numbers, aka binary numbers, we end up writing a lot of 0's and 1's and this has the potential to introduce numerous

errors. If we switch to higher base encoding, base 16, or hexadecimal numbers, we have a greater range of representations for the CPU instruction. For instance, we have digits 0 – 9 and A – F because hexadecimal numbers are base 16, we have 16 unique digits that we can use to represent an instruction's encoding. Let's revisit the fake programs we used in the previous listing, except, this time we'll use base 16 encoding.

*Listing 2-3: Fake Sample Programs in Base 16, Hexadecimal, Encoding*

<b>Program 1</b>	<b>Program 2</b>
0x00	0x0001
0x01	0x0204
0x02	0x0810
0x04	
0x08	
0x10	

A text representation of an 8 and 16-bit program. The instructions are written in hexadecimal encoding.

Things have improved a bit. The codes are shorter to write, and they are easier to read due to both the shorter text and the richer encoding of base 16. Note that it would be easier to identify errors than with the binary approach because it would cause a more noticeable difference in an instruction's encoding when compared to the binary approach. There are still a few issues, however. Let's list them here.

1. Slightly less difficult to read, trace, understand, debug.
2. Slightly more descriptive though not at all intuitive.
3. This approach is slightly less error prone.
4. No distinction between opcodes, data, and addresses.

In this case we've added a fourth reason why this approach to assembly programming is not the best. We could have applied this to the binary approach as well, but it is particularly an issue in this case. Notice that in most situations we'll use hexadecimal to represent a memory location used

in a program. This becomes an issue because with this approach to assembly programming opcodes, directives, and data all look alike. We'll have a lot of trouble tracking data and addresses with this approach.

It seems that introducing a richer encoding did us some good. We were able to use more characters to describe the instruction and it helped us read the code a bit more easily. Let's take that idea a little further with the next approach to assembly programming.

## Mnemonic Assembly Programming

In this our last approach to assembly programming, we use text, mnemonic, to represent the different instructions. We've moved away from pure numeric instructions and towards a simple, standard syntax, using text to represent the opcode, directive, data, or address and any associated arguments that are used to complete the text representation of the binary instruction.

It should stand out that this is an order of magnitude more complicated than just writing the binary or hex representation of an opcode but what we lose in simplicity we gain in readability, understanding, tracing and debugging. Let's revisit our fake programs once more. This time we'll use a fake, but similar to GenAsm, syntax/ mnemonic representation of the opcodes.

*Listing 2-4: Fake Sample Programs in Mnemonic Encoding*

### Program 1

```
MOV R1, R2  
ADD R1, R2  
CMP R2, R3  
MOV R8, R2  
MOV R9, R3  
MOV R4, R3
```

### Program 2

```
LDRB R0, [R1, #0]  
ADD R1, #1  
MOV R2, R4
```

A mnemonic representation of an 8 and 16-bit program. The instructions are fake and meant only as an example.

Now that is much better than anything we've come across thus far. We've abstracted the representation further using strings and mnemonics to refer to the opcodes, registers, and numbers used to describe an instruction. We can even infer some meaning from the short encodings used. For instance, `MOV` kind of seems like "move," and `ADD` well, that's an "add" instruction. Furthermore, aspects of the instruction like numbers and registers are much more visible. We can read the code and understand exactly what it's doing without parsing any binary or hexadecimal numbers.

On the downside we now have something which the CPU cannot understand at all. The binary and hex assembly programming approaches both created a functioning program from the start. We don't want to give up on the power we've found using text to encode our assembly program, but we also need to convert the text to the final binary program the CPU will run. That's where the assembler comes in. An assembler converts the provided assembly source code into a final binary encoding that can be run by the CPU.

Now that we've explored a few different approaches to writing a program with instruction sets and concluded that the best approach is to use mnemonic encoding, let's take a look at the assembly source code syntax used by the GenAsm assembler.

## The GenAsm Assembler Syntax

The GenAsm assembler is designed to be forward looking from a tokenization perspective. This means that you can tell the token type from the first character of the token. For instance, any string starting with an uppercase character is considered an opcode while any string starting with a lowercase character is considered a label. Let's take a look at some of the assembler's syntax rules.

**Opcodes:** Must contain only uppercase characters. An example of some opcodes are "MOV", "ADD", and "STRB."

**Directives:** Must start with the character "@" and contain only uppercase characters. Some examples are "@AREA", "@DCHW", "@EQU".

**Preprocessor Directives:** Must start with the character “\$” and contain only uppercase characters. Some examples are “\$FLIPSTRING”, “\$INCBIN”, “\$INCASM”. Preprocessor directives are handled before any other processing.

**Registers:** Registers are represented with an uppercase “R” followed by a number 0-15. Some registers also have an alias code such as PC, LR, and SP. More on this to come when we review the ARM Thumb-1 instruction set in detail.

**Register Ranges:** Register ranges can be expressed with the pattern, R#-R#, where the # characters represent ordered registers. For example, R1-R4.

**Numbers:** There are numerous ways to represent numbers in our assembly language. For instance, you can start a hex number using “0x” or “#0x”. Binary numbers can be written by prefixing the number with a “0b”. Decimal numbers can be written with the prefix character “#”.

**Labels:** Labels start with a lowercase character and can contain uppercase characters and the underscore character, “\_”.

**Comments:** Comments start with the “;” and can contain any other characters. Once a comment character is encountered all subsequent characters on the line are considered part of the comment.

**Groups and Lists:** There are two argument constructs supported. The first is the group, “[ … ]”, and the second is the list, “{ … }”. We’ll see these come up when we review the Thumb-1 instruction set in detail a little later on in the text.

**Line Syntax:** Line syntax can vary when you take into account directives and preprocessor directives. The syntax of an instruction occurs on one line and has the following general structure.

label	opcode/directive	arguments	comment
-------	------------------	-----------	---------

I should mention that individual lines of assembly code do not necessarily need to have a label or a comment. These features are used as needed and as context dictates. You couldn’t, however, have a line that starts

with a comment and has a valid opcode following it. Can you think of why that is? Well, anything following a comment is automatically a part of the comment. We'll go into much more detail regarding syntax when we start working on some test programs. Up next let's take a quick look at the listing output.

## Listing Output Format

A very important part of the assembly programming process involves a key assembler output file, the listing file. The GenAsm assembler generates both big-endian and little-endian listing and binary output for the Thumb-1 instruction set implementation. We're going to take a look at some of the important information included in the listing file. Listed subsequently is a snippet of a couple of lines from a little-endian based listing file.

*Listing 2-5: Sample from the Listing Output File*

```
1 0000000000 thumb_test ;Init Stack Pointer SP=0x03000000. Arm 2 thumb
entry point 0x00F4

2 0000000154 0134217972 0067108986 0x080000F4 00101110 01001001 0x2E 49
LDR R1, [PC, `sp_address] [184] (2e) {46}

3 0000000155 0134217974 0067108987 0x080000F6 10001101 01000110 0x8D 46
MOV SP, R1
```

A few lines of code from the GenAsm little-endian listing output file.

The first thing to notice is that there are two distinct line types in the output presented in the previous listing. The first line type, as exemplified by the first line of the example, contains no “active line” information because it is a label line with no instruction or directive, just a comment. As such this line does not have a binary representation in the linker’s output.

The second type of line present in the example is that of an active line. This type of line occurs when there is an active instruction that needs to be a part of the resulting binary file. Examples of these types of lines can be seen in lines 2 – 3 of the previous listing. In this case we have a bit more

information to express. For instance, we have a binary and hex representation of the instruction, the absolute line number, and the active line number.

Let's take a look at the break down of the data provided for an "active" line of assembly code. Again, we use the term active to denote a line of assembly source code that ends up in the final binary representation of the program.

***Listing 2-6: Breakdown of Listing Entries***

**Every Line:**

Line number absolute, int: 0000000154

**Active Line (Takes into account memory offset):**

Memory address, int: 0134217972

Line number active, int: 0067108986

Memory Address, hex: 0x080000F4

Binary representation, bin: 00101110 01001001

Hex representation, hex: 0x2E 49

Source line, string: LDR R1, [PC, `sp\_address] [184] (2e) {46}

A breakdown of the values included in the GenAsm listing file for both inactive and active lines of assembly code.

As you can see from the previous listing there is a fair amount of data about each line of assembly code present. Now of course you'll gain more experience with the listing files when you start using the GenAsm assembler to write programs. For now, it's okay just to get exposure to this information. There will be plenty of time to master it moving forward.

One last thing I'd like to mention is that there are a few extra little pieces of information injected into the listing file where a label reference is found. In the previously listed example, the `LDR` instruction is succeeded by a series of numbers and brackets. These numbers provide extra information about the value, target address, of the label reference.

The values are as follows. The first value, surrounded by square brackets `[ ]`, is a raw integer value calculated from the type of label reference.

The second value, surrounded by parenthesis { }, is a hexadecimal representation of the final value of the label reference. Lastly, the value surrounded by squiggly brackets is an integer representation of the final value of the label reference. This data will help you double check that your label references have the correct values.

The GenAsm assembler's ARM Thumb-1 instruction set implementation supports a few different types of label references. The first type is the direct address of the label and is referenced by preceding the label argument with the “=” character. The next type of label reference is the value reference. This reference is marked by preceding the label with a “~” character. This type of reference is used for value labels that are not referenced by address.

The next two label reference types are offset address references. The first is marked by preceding the label with the “-” character. This will result in a value that is the offset to the label with no adjustment for the processor pre-fetch. Lastly, the offset address reference that does take into account the processor pre-fetch is marked by preceding the label with the “`” character. Using these different label reference types, you'll be able to get the value or address you need from your labels.

That wraps up our quick review of the listing file. Don't worry if you don't feel that you've mastered this material, you'll have more exposure to it when we work on sample programs later on in the text.

## Important Topics for Assembly Programming

In this section I'll quickly cover some important topics that any assembly programmer worth their salt should be familiar with. The first topic is, tools. You're going to find that when programming in assembly you're going to be looking at memory locations and values which requires converting numbers from binary to hexadecimal to decimal and back. You're going to need a good tool. I use the following converter on my projects.

### Number Conversion Tool:

<https://www.rapidtables.com/convert/number/binary-to-hex.html>

What you'll start to realize when you get into assembly programming is that you have to be much more aware of the number base and aspects of byte encoding in ways that you might never really had to worry about when programming with higher level languages. There are a lot of tools out there for this sort of thing so just use whatever works best for you.

Next, I want to talk about some other assembly programming topics that I think are important. I've listed them subsequently and we'll talk a little bit about each one before moving on.

1. Base 2, 8, 10, 16 numbers
2. Base conversion
3. 1's complement
4. 2's complement
5. Byte order, big-endian
6. Byte order, little-endian

The first topic in the list is all about numbers. When you are working with a program in assembly you have to be able to recognize numbers of different bases and know when they should be used. For instance, if you're writing a memory location as an integer then you're probably going to kick yourself when you have to debug the memory locations on your target device, and they are all in hexadecimal number format.

Use integers where it makes sense, use hex or octal numbers for memory addresses and values, and use binary numbers for masks or other types of lower-level numeric representations. The second topic on the list follows closely behind the first one, base conversion. While I don't think anyone would expect you to work on numeric base conversions by hand you should at least have an understanding of how to do so. Again, a good tool can go a long way here. Take the time to find the right tool for your number conversion needs whether it be a calculator or online resource.

### **How to Convert Number Base:**

<https://condor.depaul.edu/psisul/conversionmath.html>

The next two entries in the previous listing are the 1's and 2's complement encoding. One's and two's complements are used in computers

to handle representing negative numbers. The encodings are also used to handle arithmetic of negative numbers. While you won't usually have to deal with these concepts too much while writing assembly programs you will encounter them when you build an assembler so let's touch upon the topic here.

### **Ones Complement Explained:**

[https://en.wikipedia.org/wiki/Ones%27\\_complement](https://en.wikipedia.org/wiki/Ones%27_complement)

Remember we're building an assembler but we're also using it to implement the ARM Thumb-1 instruction set. I want to make sure this distinction is clear. To accomplish this task, we're going to have to become Thumb-1 experts and to a lesser extent be aware of assembly programming topics like numeric encodings and negative number representation.

The one's complement of a binary number is the value obtained by inverting all the bits in the binary representation of the number. Inversion in this sense means swapping 0's and 1's. This mathematical operation is used in some cases to represent negative numbers. It also provides a system, one's complement arithmetic, in which negative numbers are represented by the 1's complement of the corresponding positive number. An N-bit 1's complement numeral system can only represent integers in the range  $-(2^{(N-1)} - 1)$  to  $2^{(N-1)} - 1$ .

Taking the 1's complement is a useful technique for representing negative numbers in computing and in some cases, you may need to take the 1's complement of a numeric argument to a directive or opcode. That brings us to the next encoding, the 2's complement. Similar to the 1's complement, the 2's complement is a mathematical operation performed on binary numbers. It is used in computing as a method of representing signed integers and a system of arithmetic that handles negative numbers.

### **Two's Complement Explained:**

<https://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html>

The 2's complement of an N-bit number is defined as its complement with respect to  $2^N$ ; the sum of a number and its 2's complement is  $2^N$ . For example, given the 3-bit number 010, the 2's complement is 110, because

$010 + 110 = 1000 = 2^3$ . The 2's complement is calculated by inverting the bits and adding one.

This encoding is the most common technique for representing signed integers on computers. The range of the 2's complement is slightly better than that of the 1's complement and can represent integers in the range  $-(2^{(N-1)})$  to  $2^{(N-1)} - 1$ . A convenient way of calculating the 2's complement is to take the 1's complement and add 1. The reason for this is because the sum of a number and its 1's complement is all 1-bits or  $2^N - 1$ ; and the sum of a number and its 2's complement is  $2^N$ . Thus  $2^N - 1 + 1 = 2^N$  and we have arrived at the 2's complement from the 1's complement.

In any case, as an astute software developer writing an assembler, or as a master assembly programmer, you want to be aware of the different encodings used to represent numbers and at least have a working familiarity with them. The last two topics for us to discuss have to do with the concept of endianness. This concept refers to the order of the bytes used to represent a number.

In one case the smaller bits are on the left and the larger bits are on the right, big-endian. In the other case the larger bits are on the left and the smaller bits are on the right, little-endian. We'll encounter these topics on a case-by-case basis as they come up throughout the GenAsm assembler's detailed code review.

## Chapter Conclusion

In this chapter we covered a lot of material. Let's take a look at the topics we've discussed and provide a little more detail about them below.

- 1. Assembly Programming Fundamentals:** In this section we took a look at the core aspects of assembly programming. We talked about what constitutes a computer program and how a CPU processes said program.
- 2. Binary Assembly Programming:** In this section we took a look at the possibility of programming CPU instructions directly in binary format.

3. **Hex Assembly Programming:** To belabor the point of programming a CPU directly using the numeric representation of an instruction we also discussed the pros and cons of doing so in hexadecimal numeric encoding as opposed to binary numeric encoding.
4. **Mnemonic Assembly Programming:** Lastly, wrapping up the sections on assembly programming we introduced the concept of mnemonic assembly programming in which case we use simple string codes to represent the CPU instructions and their arguments.
5. **The GenAsm Assembler Syntax:** In this section we took a crash course in assembler syntax and took a look at the syntax used to implement the ARM Thumb-1 instruction set in the GenAsm extensible assembler software.
6. **Listing Output Format:** This section was all about one of the most important assembler outputs, the listing file. We took a look at an example snippet from a listing file and reviewed the different pieces of information the liker adds to the assembly source when generating the listing file.
7. **Important Topics for Assembly Programming:** We wrapped up the chapter with a more general section on some important assembly programming topics. We covered the need to have some good tools when it comes to converting numbers etc. We also covered some important topics that come up in assembly programming like base, complements, and endianness.

This chapter really sets the stage for us and gives us some key foundational knowledge we'll need as we explore how an assembler is written in Java and how an assembler is able to convert mnemonic CPU instruction representations to binary output. From this point on in the text you will be guided on a journey to learn the intricacies of implementing an assembler in a line-by-line code review. What is more, you'll actually use the assembler to write, assemble, and run test programs that run on a GameBoy Advance emulator.



# Chapter 3: ARM Thumb-1 Opcodes Part 1

We've begun our journey into the world of building assemblers by introducing some basic assembly programming concepts. We've also had a chance to take a look at the expected syntax of the GenAsm assembler's ARM Thumb-1 instruction set implementation. It's important to realize that we're not only talking about creating a simple, extensible, programming API that allows us to implement an assembler. We're also talking about implementing an instruction set and building sample programs.

We just happen to be implementing the ARM Thumb-1 instruction set as our first assembly language. I want to mention that information on each opcode group is gathered from the **ARM7TDMI** data sheet, version **DDI-0029E**. I make no claim on the text from this document. I'm presenting a re-formatted, adjusted version of the instruction definitions for educational purposes. I've made adjustments and added information where it applies within the context of this project.

You can find a copy of the documentation online by searching for "ARM7TDMI DDI-0029E" in your favorite search engine. I've also included a copy of the documentation used with the GenAsm project. It is located in the NetBeans project's local directory at this location, ".\storage\

Don't worry, we're going to revisit the assembler syntax and listing file output, in detail, when we start building and reviewing the Thumb-1 sample programs that we'll use to put our assembler through its paces. We'll also cover the programming aspect of the project, in detail, as we cover the Java classes and methods that power the GenAsm assembler.

Well, if we're going to build an assembler, we'd better have an instruction set in mind to support. As I've mentioned before we'll be using the ARM Thumb-1 instruction set and without further ado, let me introduce you to it.

## The ARM Thumb-1 Instruction Set

ARM, Advanced RISC Machines, is a family of reduced instruction set computing (RISC) architectures for computer processors, configured for various environments. ARM develops the architecture and licenses it to other companies, who design their own products that implement one of those architectures. It also designs cores that implement the instruction set and licenses these designs to a number of companies that incorporate those core designs into their own products.

ARM has designed a few different instruction sets. There's the A64, A32, T32 and the T16 subset, which is also referred to as the Thumb-1 instruction set, that we'll focus on in this text. The Thumb-1 instruction set is a subset of the most commonly used 32-bit ARM instructions. Thumb-1 instructions are each 16 bits long and correspond to a 32-bit ARM instruction that takes the same action on the processor. Thumb-1 instructions operate with the standard ARM register configuration, allowing excellent interoperability between ARM and Thumb-1 states on the same chip.

Upon execution, Thumb-1 16-bit instructions are transparently decompressed to full 32-bit ARM instructions in real time with no performance loss. The Thumb-1 instruction set has the advantage of a 32-bit core:

1. 32-bit address space
2. 32-bit registers

3. 32-bit shifter, and arithmetic logic unit (ALU)
4. 32-bit memory transfer

The Thumb-1 instruction set offers a long branch range, powerful arithmetic operations, a large address space, and a very dense binary footprint with Thumb-1 code being on average 65% smaller than code written with a 32-bit ARM instruction set. For this reason, and because the overall size of the Thumb-1 instruction set is relatively small, containing around 36 opcodes, we chose to work with the Thumb-1 instruction set for this project.

A note about instruction cycle times. Due to the nature of this text and the inherent limitations therein I've decided that focusing on opcode execution cycles is outside of the scope of this book. If you want to accurately track the cycles needed to execute instructions, opcodes, I recommend extending the current implementation to handle instruction cycle calculations.

The registers used by the Thumb-1 instruction set are separated into low and high registers. In each case a register is referenced with a 3-bit value. This means that low and high registers are described in the same way, using a 3-bit value, with a range of 0 to 7. It's the context of the opcode that indicates which type of register you're referring to. Furthermore, there are a few aliases for some of the special high registers. I've listed them as follows.

*Listing 3-1: ARM Thumb-1 Instruction Set Registers*

Mnemonic	Alias	Binary Form	Description
R0	-	000	Standard low range register.
R1	-	001	Standard low range register.
R2	-	010	Standard low range register.
R3	-	011	Standard low range register.
R4	-	100	Standard low range register.
R5	-	101	Standard low range register.

R6	-	110	Standard low range register.
R7	-	111	Standard low range register.
R8	-	000	Standard hi range register.
R9	-	001	Standard hi range register.
R10	-	010	Standard hi range register.
R11	-	011	Standard hi range register.
R12	-	100	Standard hi range register.
R13	SP	101	The SP register is the stack pointer register and it's set to point to the first memory location of the program's stack.
R14	LR	110	The LR register is the link register and it's used to hold the return address for a function call.
R15	PC	111	The PC register is the program counter register and it's used to hold the memory address of the next instruction to execute. Due to prefetch this value is ahead by 4 bytes.
CPSR	-	-	The Current Program Status Register (CPSR) holds processor status and control information that is set upon the execution of certain opcodes.

A listing of the registers available on the ARM7TDMI CPUs we'll be working with.

We won't touch upon the CPSR register in too much detail throughout the text. This is because the register is mainly managed by the instruction set architecture and is updated after certain opcodes are executed. We will, however, gain some experience with it when we work on writing some test programs for our assembler. To be thorough I'll list the bit breakdown of the CPSR register here.

*Listing 3-2: ARM Instruction Set CPSR Register Bit Breakdown*

Bits	Name	Function
31	N	Negative condition code flag
30	Z	Zero condition code flag
29	C	Carry condition code flag
28	V	Overflow condition code flag
27	Q	Cumulative saturation bit
26-25	IT [1-0]	If-Then execution state bits for the Thumb IT (If-Then) instruction
24	J	Jazelle bit
19-16	GE	Greater than or Equal flags
15-10	IT [7-2]	If-Then execution state bits for Thumb IT (If-Then) instruction
9	E	Endianness execution state bit: 0 = Little-endian, 1 = Big-endian
8	A	Asynchronous abort mask bit
7	I	IRQ mask bit
6	F	FIRQ mask bit
5	T	Thumb execution state bit
4-0	M	Mode field

A bit level breakdown of the CPSR register.

We won't be interacting directly with the CPSR bits frequently, but they will come up in the test programs later on in the text. We'll take a look at the

JSON format we'll use to store the opcode instruction's details in the next section.

## The Thumb-1 Opcode's Structure

I want to quickly review the structure of a Thumb-1 opcode as we'll review it in this text. In general, when you're dealing with any opcodes, data, or memory addresses in Java, and in the GenAsm assembler program, you're using big-endian byte format. As part of the GenAsm assembler both a little- and big-endian listing and binary output files are generated.

We'll look at each Thumb-1 opcode in turn, in big-endian format and we'll take a look at the JSON object that is used to describe the opcode to the GenAsm program. This means that we're not only looking over the Thumb-1 opcodes but also how the data is stored such that it can be used by the GenAsm assembler's Thumb-1 implementation. When we discuss the opcodes, we'll use the following big-endian bit breakdown. The largest weighted bits are on the left, big end first, and the smallest bits are on the right.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
-----															

An example of the bit listing used to describe a Thumb-1 opcode instruction is as follows. The subsequent example is of the `LSL`, `LSR`, and `ASR` opcode group. Fair warning, I use the terms "opcode" and "instruction" somewhat interchangeably. This is probably not a good habit but as long as you understand the context, you'll understand what's going on.

Opcodes are really the mnemonic referenced by the instruction and not the actual individual instruction as there can be a group of resulting instructions associated with the same opcode. Similarly, an opcode group or instruction group is a set of opcodes that share a similar structure with regard to their binary encoding, and subsequently, their assembly syntax. Again, just make sure you understand the context and you'll be fine.

# Format-1: Move Shifted Register - LSL, LSR, ASR

## Bit Breakdown: Move Shifted Register

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
-----															
0   0   0   Op          Offset5          Rs          Rd															

## Details:

Op = Opcode

Op = 0 → Opcode = LSL

Op = 1 → Opcode = LSR

Op = 2 → Opcode = ASR

Offset5 = 5-bit immediate value

Rs = Source register, low

Rd = Destination register, low

Cond. Codes = Condition codes are set by this opcode group.

Notes = These instructions move a shifted value between Lo registers.

## Description:

Op	Thumb-1 Code	Action
00	LSL Rd, Rs, #Offset5	Shift Rs left by a 5-bit immediate value and store the results in Rd.
01	LSR Rd, Rs, #Offset5	Perform a logical shift right on Rs by a 5-bit immediate value and store the results in Rd.
02	ASR Rd, Rs, #Offset5	Perform an arithmetic shift right on Rs by a 5-bit immediate value and store the result in Rd.

## Examples:

Endian	Code	Binary Rep	Hex Rep
Little	LSL R0, R1, #0	00001000 00000000	0x08 00
Big	LSL R0, R1, #0	00000000 00001000	0x00 08

Little	LSR R0, R1, #0	00001000 00001000	0x08 08
Big	LSR R0, R1, #0	00001000 00001000	0x08 08
Little	ASR R0, R1, #31	11001000 00010111	0xC8 17
Big	ASR R0, R1, #31	00010111 11001000	0x17 C8

#### JSON Representation:

```
{
  "obj_name": "is_op_code",
  "op_code_name": "LSL",
  "index": 0,
  "arg_len": 3,
  "arg_separator": ",",
  "bit_rep": {"bit_string": "00000", "bit_int": 0, "bit_len": 5},
  "bit_series": {"bit_start": 11, "bit_stop": 15, "bit_len": 5},
  "args": [
    {
      "obj_name": "is_op_code_arg",
      "arg_index": 0,
      "bit_index": 0,
      "is_entry_types": ["RegisterLow"],
      "is_arg_type": "Rd",
      "bit_series": {"bit_start": 0, "bit_stop": 2, "bit_len": 3}
    },
    {
      "obj_name": "is_op_code_arg",
      "arg_index": 1,
      "bit_index": 1,
      "is_entry_types": ["RegisterLow"],
      "is_arg_type": "Rs",
      "bit_series": {"bit_start": 3, "bit_stop": 5, "bit_len": 3}
    },
    {
      "obj_name": "is_op_code_arg",
      "arg_index": 2,
      "bit_index": 2,
      "is_entry_types": ["Number", "LabelRef"],
      "is_arg_type": "Offset5",
      "bit_series": {"bit_start": 6, "bit_stop": 10, "bit_len": 5}
    }
  ]
}
```

```

        "bit_series": {"bit_start": 6, "bit_stop": 10, "bit_len": 5},
        "num_range": {"min_value": 0, "max_value": 31, "bit_len": 5},
    "twos_complement": false, "alignment": "NATURAL"},
        "bit_shift": {"shift_dir": "NONE", "shift_amount": 0}
    }
]
}

```

For the most part the opcode listing is straightforward. As we review each instruction, we'll review anything new that we haven't seen before. As such the further we go into the instruction set the less dialogue we'll have. This is fine. You need to have some exposure to an instruction set if you're going to build an assembler and this is a good way to start that process. There are a few new concepts for us to cover right out of the gate.

For instance, the Thumb-1 implementation in the GenAsm assembler is such that all the instruction set data is stored in a series of JSON data files. The file that holds all the opcode information has entries for each code that describe how to construct the opcode both in text, assembly code, form and in binary form. Let's review the opcode entry, section by section. The first section is the bit breakdown. This section describes the format of the 16-bit opcode bit by bit. Some bit values are static, for instance the bits that represent the given opcode, while some represent certain types that are used throughout the instruction set like a register, `Rn`, or an immediate value, `Offset5`.

Note the use of the `Op` type in the bit breakdown. Often you will see opcodes that are overloaded to perform similar operations. In this case the `Op` type in the bit breakdown determines if the CPU runs an `LSL`, `LSR` or `ASR` action. In this case the `Op` type is not an argument expressed in assembly code, it's an encoding handled by the assembler. It may be a little difficult at first to determine what responsibilities the assembler has and what responsibilities lie with the instruction set or the programmer. This will become clearer as you gain experience working with building assemblers and writing programs in assembly.

The details section provides extra information about each type used to describe the opcode. In this case we describe the use of the `Op` types, the `Offset5` value, the `Rd` and `Rs` registers, and there's a note about the condition codes at the end of the section. Condition codes indicate certain key

aspects of the last CPU action. For instance, it contains flags that indicate if the last two values compared were equal, if a zero has been encountered, if there was an overflow during arithmetic, etc.

These flags are used for certain logic operation in assembly programming, but we'll get more exposure to them later on in the text. For now, let's direct our attention to the next section of an opcode entry's definition. The description section is used to describe the opcode in use with generic arguments. In this case an entry for the three permutations of this opcode are present. Some descriptive text is also listed to provide some depth to the described opcodes.

In the next section, examples, there are a few examples of the opcode in use in assembly code with the general arguments replaced by actual values. Each example has its associated binary output listed alongside it for both little- and big-endian byte encodings. The last section is perhaps the most erudite upon initial inspection. Fear not, you'll understand all about it in no time!

In order to drive the GenAsm assembler, such that we can process the assembly code for a target instruction set, we need to have the instruction set details stored in data files so that the assembler can access and utilize the data. For our purposes we chose the JSON format because it's simple, lightweight, and it can model objects and simple hierarchies fairly well. It also has support in many languages and in many contexts.

In short, it's perfect for storing this kind of data and it's extremely portable so we don't have to worry about finding parsing libraries. The JSON representation shows both the assembly code syntax but also the bit breakdown of the opcode. To accomplish this there are some special JSON objects, sub-objects/child objects, used to describe different aspects of the opcode's entry. Because this is our first encounter with the JSON representation of an opcode entry, we'll cover it in detail.

The first field, `obj_name`, is a constant value and is always set to the name of the object that this JSON structure represents. In this case the name of the object is, "is\_op\_code." The next field, `op_code_name`, is the mnemonic used to represent this opcode in assembly source code. Now there may be multiple opcode entries with the same `op_code_name` field value.

This is because there are often a few different ways an opcode can be called. To overcome this the assembler will match opcodes by their mnemonic and their argument signature to ensure uniqueness.

Next up we have the `index` field. This field represents the unique index of this opcode entry in the set of supported opcodes. The `arg_length` field describes the number of arguments this permutation of the current opcode takes. The `next` field describes how to represent this opcode in binary form. The `bit_rep` field is a bit representation object that contains the binary encoding for this opcode, in this case “00000”, as well as the integer value of the binary representation and an expected length.

Subsequently there’s the `bit_series` field. This field holds a bit series object that is used to describe the position in the bit breakdown of the opcode’s binary encoding, “00000”. In this case the binary encoding starts at bit 11, zero based index. Ends at bit 15 and has a length of 5 bits. That perfectly describes where to place the binary encoding for this opcode, “00000”, in the resulting binary encoding for this line of assembly code. Lastly, we have the `arg_separator` field. This field holds the character that should be used to separate out the opcode arguments. The last field of the main object is the `args` field. This field is used to hold an array of object entries that describe the opcode’s arguments.

The object entries are all of type “`is_op_code_arg`” as noted by the `obj_name` field of each argument object. Almost all of the objects contained in JSON data files have the `obj_name` field populated. Some of the smaller objects like `bit_series` and `bit_rep` do not have this field due to the simplicity of the object and for brevity’s sake. The first two argument entries describe the register opcode arguments `Rd` and `Rs`. Note the `arg_index` and `bit_index` fields. These fields are used to represent order in the assembly code syntax and the binary representation of said assembly code. In other words, these fields track this argument’s position in the assembly source code and its position in the binary representation of the opcode. Let’s list the opcode argument JSON data here.

*Listing 3-3: The Opcode Argument JSON Data*

```
{  
  "obj_name": "is_op_code_arg",
```

```

    "arg_index": 0,
    "bit_index": 0,
    "is_entry_types": ["RegisterLow"],
    "is_arg_type": "Rd",
    "bit_series": {"bit_start": 0, "bit_stop": 2, "bit_len": 3}
},
{
    "obj_name": "is_op_code_arg",
    "arg_index": 1,
    "bit_index": 1,
    "is_entry_types": ["RegisterLow"],
    "is_arg_type": "Rs",
    "bit_series": {"bit_start": 3, "bit_stop": 5, "bit_len": 3}
},
{
    "obj_name": "is_op_code_arg",
    "arg_index": 2,
    "bit_index": 2,
    "is_entry_types": ["Number", "LabelRef"],
    "is_arg_type": "Offset5",
    "bit_series": {"bit_start": 6, "bit_stop": 10, "bit_len": 5},
    "num_range": {"min_value": 0, "max_value": 31, "bit_len": 5},
    "twos_complement": false, "alignment": "NATURAL",
    "bit_shift": {"shift_dir": "NONE", "shift_amount": 0}
}

```

A listing of the LSR opcode's arguments in JSON format.

The next field in the argument object is the `is_entry_types` field. This field is very important as it describes the type of objects that are allowed to be used to fill this argument requirement. In this case things are simple, the only object type allowed here is a low register, registers 0 – 7. The next field, `is_arg_type`, is a string that represents the way this opcode argument is referred to in the bit breakdown. In this case the argument is named `Rd`. The last field in this type of argument object is the `bit_series` field. This field is similar to the previous `bit_series` field we encountered in the opcode object. It describes the position and length of this argument shown in the bit breakdown.

You've noticed that the first two argument objects are very similar but the third one, however, is not. Different argument objects will have different field values filled in depending on the type of data they represent. For instance, the third argument in the previously listed series of opcode argument objects is used to represent an `Offset5` data type as opposed to a `Rd`, `Rs` register data type. Note that the argument can be satisfied in different ways depending on the `is_entry_types` field value.

Take a look at the `is_entry_type` field of the third argument. It takes a `Number` or `LabelRef` object, the value of which must be converted by the assembler into an `Offset5`, 5-bit binary representation. Again, the main takeaway here is that there are different object types that can satisfy an opcode argument requirement. This argument object has the same fields as those used to describe register arguments but with two more fields. The `num_range` and `bit_shift` fields.

The `num_range` object is used to describe the valid range of values for the given opcode argument while the `bit_shift` field is used to describe any bit shifting the assembler must perform before committing the argument value to binary encoding. That wraps up our review of the opcode entry's structure including a detailed review of the JSON data representation of this opcode. As we explore further into the instruction set there will be fewer new JSON objects and other topics of discussion.

As such the remainder of the opcode entries will feel more like a reference or documentation. This is fine. The opcode listing does serve as a reference, but I want you to carefully read through each opcode's entry and try to imagine using it in some kind of assembly program. It's okay if you still don't have any idea how to connect different opcodes to make a program. That part comes up a bit further down the road in this text. After all, we still have to build our assembler!

## Thumb-1 Opcodes Part 1

Picking up right where we left off, we'll continue our review of the instruction set with the opcode directly following the `LSL`, `LSR`, `ASR` group.

## Format-2: Add/Subtract - ADD, SUB

### Bit Breakdown: Add/Subtract

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
-----															
0   0   0   1   1   I   Op   Rn/Offset3   Rs     Rd															

### Details:

I = Immediate flag

I = 0 → Rn, register operand

I = 1 → Offset3, immediate operand

Op = Opcode

Op = 0 → Opcode = ADD

Op = 1 → Opcode = SUB

Offset3 = 3-bit immediate value

Rs = Source register, low

Rd = Destination register, low

Cond. Codes = Condition codes are set by this opcode group.

Notes = These instructions allow the contents of a Lo register of 3-bit immediate value to be added or subtracted from a Lo register.

### Description:

Op	I	Thumb-1 Code	Action
0	0	ADD Rd, Rs, Rn	Add contents of Rn to contents of Rs. Place in Rd.
0	1	ADD Rd, Rs, #Offset3	Add 3-bit immediate value to contents of Rs. Place result in Rd.
1	0	SUB Rd, Rs, Rn	Subtract contents of Rn from contents of Rs. Place result in Rd.
1	1	SUB Rd, Rs, #Offset3	Subtract 3-bit immediate value from contents of Rs. Place result in Rd.

### Examples:

Endian	Code	Binary Rep	Hex Rep
--------	------	------------	---------

Little	ADD R0, R1, R2	10001000 00011000	0x88 18
Big	ADD R0, R1, R2	00011000 10001000	0x18 88
Little	ADD R0, R1, #7	11001000 00011101	0xC8 1D
Big	ADD R0, R1, #7	00011101 11001000	0x1D C8
Little	SUB R0, R1, R2	10001000 00011010	0x88 1A
Big	SUB R0, R1, R2	00011010 10001000	0x1A 88

#### JSON Representation:

```
{
  "obj_name": "is_op_code",
  "op_code_name": "ADD",
  "index": 3,
  "arg_len": 3,
  "arg_separator": ",",
  "bit_rep": {"bit_string": "0001100", "bit_int": 12, "bit_len": 7},
  "bit_series": {"bit_start": 9, "bit_stop": 15, "bit_len": 7},
  "args": [
    {
      "obj_name": "is_op_code_arg",
      "arg_index": 0,
      "bit_index": 0,
      "is_entry_types": ["RegisterLow"],
      "is_arg_type": "Rd",
      "bit_series": {"bit_start": 0, "bit_stop": 2, "bit_len": 3}
    },
    {
      "obj_name": "is_op_code_arg",
      "arg_index": 1,
      "bit_index": 1,
      "is_entry_types": ["RegisterLow"],
      "is_arg_type": "Rs",
      "bit_series": {"bit_start": 3, "bit_stop": 5, "bit_len": 3}
    },
    {
      "obj_name": "is_op_code_arg",
      "arg_index": 2,
      "bit_index": 2,
      "is_entry_types": ["RegisterLow"],
      "is_arg_type": "Rd",
      "bit_series": {"bit_start": 6, "bit_stop": 8, "bit_len": 3}
    }
  ]
}
```

```

        "arg_index": 2,
        "bit_index": 2,
        "is_entry_types": ["RegisterLow"],
        "is_arg_type": "Rn",
        "bit_series": {"bit_start": 6, "bit_stop": 8, "bit_len": 3}
    }
]
},
{
    "obj_name": "is_op_code",
    "op_code_name": "ADD",
    "index": 4,
    "arg_len": 3,
    "arg_separator": ",",
    "bit_rep": {"bit_string": "0001110", "bit_int": 14, "bit_len": 7},
    "bit_series": {"bit_start": 9, "bit_stop": 15, "bit_len": 7},
    "args": [
        {
            "obj_name": "is_op_code_arg",
            "arg_index": 0,
            "bit_index": 0,
            "is_entry_types": ["RegisterLow"],
            "is_arg_type": "Rd",
            "bit_series": {"bit_start": 0, "bit_stop": 2, "bit_len": 3}
        },
        {
            "obj_name": "is_op_code_arg",
            "arg_index": 1,
            "bit_index": 1,
            "is_entry_types": ["RegisterLow"],
            "is_arg_type": "Rs",
            "bit_series": {"bit_start": 3, "bit_stop": 5, "bit_len": 3}
        },
        {
            "obj_name": "is_op_code_arg",
            "arg_index": 2,
            "bit_index": 2,
            "is_entry_types": ["Number", "LabelRef"],
            "is_arg_type": "Offset3",
            "bit_series": {"bit_start": 6, "bit_stop": 8, "bit_len": 3},
            "num_range": {"min_value": 0, "max_value": 7, "bit_len": 3},
            "twos_complement": false, "alignment": "NATURAL"},
            "bit_shift": {"shift_dir": "NONE", "shift_amount": 0}
        }
    ]
}

```

```
        }  
    ]  
}
```

Because the ADD and SUB opcodes in this group have a few too many permutations to list I've limited the examples to only six entries. The missing entries are very similar to the ones listed save for their binary and hexadecimal representation. Similarly, I've only listed two of the four JSON data entries. Due to their similarity listing all four would be redundant.

The first thing I'd like to point out is that the JSON data entry represents one, unique, permutation of an opcode and its arguments. For instance, in this case the ADD and SUB opcodes each have two ways they can be expressed. One which uses only registers and a second method which uses registers and a small immediate value. The main point of discussion with regard to this JSON data listing is the Offset3 argument type in the second JSON object. Did you notice that the is\_entry\_types field has the same value as the Offset5 argument type we looked at earlier? Any ideas why that is?

The way the Thumb-1 instruction set was implemented in the GenAsm assembler, the software converts the source number, either from a number typed into the assembly source code or from a label which is a reference to a value or an address. All three forms of information are numbers and as long as the number fits into the valid range of the Offset3 argument any which one will do.

In this way the assembler takes care of converting different token types, Number, LabelRef, etc., into the data type defined by the instruction set. Note that the bit series and number range sub-objects of the Offset3 argument object are similar but slightly different to that of the Offset5 based argument we reviewed earlier. In general, just because an argument is of a certain instruction set type doesn't mean that it is exactly the same as others you've seen previously. Some opcodes require further adjustment to the Offset3 or Offset5 number before the entry can be properly represented as a binary encoding.

Please note that some opcodes will appear multiple times in different groups due to different use cases. An example would be load and store opcodes. There are different ways to describe what information to load and

where to load it from. The same goes for the store opcodes. I know this can feel like a bit much to take in at first, but you'll see that over time it will become second hand.

## **Format-3: Move/Compare/Add/Subtract Immediate - MOV, CMP, ADD, SUB**

### **Bit Breakdown: Move/Compare/Add/Subtract**

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
<hr/>															
0   0   1   Op     Rd       Offset8															

### **Details:**

Op = Opcode

Op = 0 = 00 → Opcode = MOV

Op = 1 = 01 → Opcode = CMP

Op = 2 = 10 → Opcode = ADD

Op = 3 = 11 → Opcode = SUB

Offset8 = 8-bit immediate value

Rd = Destination register, low

Cond. Codes = Condition codes are set by this opcode group.

Notes = The instructions in this group perform operations between a Lo register and an 8-bit immediate value.

### **Description:**

Op	Thumb-1 Code	Action
0	MOV Rd, #Offset8	Move 8-bit immediate value into Rd.
00		
1	CMP Rd, #Offset8	Compare contents of Rd with 8-bit immediate value.
01		
2	ADD Rd, #Offset8	Add an 8-bit immediate value to the contents of Rd and place the result in Rd.
10		
3	SUB Rd, #Offset8	Subtract an 8-bit immediate value from the contents of Rd and place the
11		

result in Rd.

**Examples:**

Endian	Code	Binary Rep	Hex Rep
Little	MOV R0, #0	00000000 00100000	0x00 20
Big	MOV R0, #0	00100000 00000000	0x20 00
Little	CMP R0, #255	11111111 00101000	0xFF 28
Big	CMP R0, #255	00101000 11111111	0x28 FF
Little	ADD R0, #255	11111111 00110000	0xFF 30
Big	ADD R0, #255	00110000 11111111	0x30 FF

**JSON Representation:**

```
{  
    "obj_name": "is_op_code",  
    "op_code_name": "MOV",  
    "index": 7,  
    "arg_len": 2,  
    "arg_separator": ",",  
    "bit_rep": {"bit_string": "00100", "bit_int": 4, "bit_len": 5},  
    "bit_series": {"bit_start": 11, "bit_stop": 15, "bit_len": 5},  
    "args": [  
        {  
            "obj_name": "is_op_code_arg",  
            "arg_index": 0,  
            "bit_index": 1,  
            "is_entry_types": ["RegisterLow"],  
            "is_arg_type": "Rd",  
            "bit_series": {"bit_start": 8, "bit_stop": 10, "bit_len": 3}  
        },  
        {  
            "obj_name": "is_op_code_arg",  
            "arg_index": 1,  
            "bit_index": 0,  
            "is_entry_types": ["RegisterLow"],  
            "is_arg_type": "Rd",  
            "bit_series": {"bit_start": 11, "bit_stop": 15, "bit_len": 5}  
        }  
    ]  
}
```

```

        "bit_index": 0,
        "is_entry_types": ["Number", "LabelRef"],
        "is_arg_type": "Offset8",
        "bit_series": {"bit_start": 0, "bit_stop": 7, "bit_len": 8},
        "num_range": {"min_value": 0, "max_value": 255, "bit_len": 8},
    "twos_complement": false, "alignment": "NATURAL"},
        "bit_shift": {"shift_dir": "NONE", "shift_amount": 0}
    }
]
}

```

One new aspect of the instruction set in this group of opcodes is the `Offset8` argument data type. This is another type of numeric argument that has a range of 0 to 255 as listed previously. Note that the `bit_series` object reflects the size of the `Offset8` date type and that there is no bit shift applied to this numeric value. I have only listed six examples, as will most likely be the case when there are too many, which excludes a few. The assembly code is very similar, but the binary encoding will be different. Take a moment to work through some examples on your own.

Similarly, I've only listed one JSON opcode entry, the entry for this opcode group's `MOV` instruction. The other data entries are very similar so there is no need to list them here. You can look up any opcode's JSON data entry in the file, `is_op_codes.json`, found at the following location relative to the GenAsm NetBeans project folder.

`./cfg/THUMB/is_op_codes.json`

On to the next set of opcodes!

## Format-4: ALU Operations

### Bit Breakdown: ALU Operations

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
-----															
0   1   0   0   0   0   0   Op     Rs     Rd															

### Details:

Op = Opcode  
 $Op = 00 = 0000 \rightarrow \text{Opcode} = \text{AND}$

Op = 01 = 0001 -> Opcode = EOR  
 Op = 02 = 0010 -> Opcode = LSL  
 Op = 03 = 0011 -> Opcode = LSR  
 Op = 04 = 0100 -> Opcode = ASR  
 Op = 05 = 0101 -> Opcode = ADC  
 Op = 06 = 0110 -> Opcode = SBC  
 Op = 07 = 0111 -> Opcode = ROR  
 Op = 08 = 1000 -> Opcode = TST  
 Op = 09 = 1001 -> Opcode = NEG  
 Op = 10 = 1010 -> Opcode = CMP  
 Op = 11 = 1011 -> Opcode = CMN  
 Op = 12 = 1100 -> Opcode = ORR  
 Op = 13 = 1101 -> Opcode = MUL  
 Op = 14 = 1110 -> Opcode = BIC  
 Op = 15 = 1111 -> Opcode = MVN

Rs = Source register, low

Rd = Destination register, low

Cond. Codes = Condition codes are set by this opcode group.

Notes = The instructions in this opcode group perform ALU operations on a Lo register pair.

#### Description:

Op	Thumb-1 Code	Action
00	AND Rd, Rs	Rd = Rd AND Rs
0000		
01	EOR Rd, Rs	Rd = Rd EOR Rs
0001		
02	LSL Rd, Rs	Rd = Rd << Rs
0010		
03	LSR Rd, Rs	Rd = Rd >> Rs
0011		
04	ASR Rd, Rs	Rd = Rd ASR Rs
0100		
05	ADC Rd, Rs	Rd = Rd + Rs + C-bit (CPSR carry flag)
0101		

06 0110	SBC Rd, Rs	Rd = Rd + Rd + NOT C-bit (CPSR carry flag)
07 0111	ROR Rd, Rs	Rd = Rd ROR Rs
08 1000	TST Rs, Rs	Set condition codes on Rd and Rs.
09 1001	NEG Rd, Rs	Rd = -Rs
10 1010	CMP Rd, Rs	Set condition codes on Rd - Rs.
11 1011	CMN Rd, Rs	Set condition codes on Rd + Rs.
12 1100	ORR Rd, Rs	Rd = Rd OR Rs
13 1101	MUL Rd, Rs	Rd = Rs * Rd
14 1110	BIC Rd, Rs	Rd = Rd AND NOT Rs
15 1111	MVN Rd, Rs	Rd = NOT Rs

**Examples:**

Endian	Code	Binary Rep	Hex Rep
Little	AND Rd, Rs	00001000 01000000	0x08 40
Big	AND Rd, Rs	01000000 00001000	0x40 08
Little	ADC Rd, Rs	01001000 01000001	0x48 41
Big	ADC Rd, Rs	01000001 01001000	0x41 48

Little	MVN Rd, Rs	11001000 01000011	0xC8 43
Big	MVN Rd, Rs	01000011 11001000	0x43 C8

#### JSON Representation:

```
{
    "obj_name": "is_op_code",
    "op_code_name": "CMP",
    "index": 21,
    "arg_len": 2,
    "arg_separator": ",",
    "bit_rep": {"bit_string": "0100001010", "bit_int": 266, "bit_len": 10},
    "bit_series": {"bit_start": 6, "bit_stop": 15, "bit_len": 10},
    "args": [
        {
            "obj_name": "is_op_code_arg",
            "arg_index": 0,
            "bit_index": 0,
            "is_entry_types": ["RegisterLow"],
            "is_arg_type": "Rd",
            "bit_series": {"bit_start": 0, "bit_stop": 2, "bit_len": 3}
        },
        {
            "obj_name": "is_op_code_arg",
            "arg_index": 1,
            "bit_index": 1,
            "is_entry_types": ["RegisterLow"],
            "is_arg_type": "Rs",
            "bit_series": {"bit_start": 3, "bit_stop": 5, "bit_len": 3}
        }
    ]
}
```

There are a lot of opcodes in this group, but they all follow the same general pattern so we'll only list one example of the JSON object. Similarly, there are a lot of examples for the opcodes in this group. I've only listed six examples here, but you can find more in the output files for test program A. The following output files are useful when finding examples of an opcode in use and can be found at the location shown here.

```
./cfg/THUMB/OUTPUT/TEST_A_AllOpCodes\
output_assembly_listing_endian_lil.list
```

```
./cfg/THUMB/OUTPUT/TEST_A_AllOpCodes\  
output_assembly_listing_endian_big.list
```

Also due to the redundant nature of the JSON objects I've only listed one such example, previously, the `CMP` opcode. As I mentioned before all opcodes in this group are very similar so listing all of them here would be redundant. However, if you need to look up the JSON data entry for a particular opcode by checking the `is_op_codes.json` data file as mentioned earlier.

## Format-5: Hi Register Operations/Branch Exchange - ADD, CMP, MOV, BX

### Bit Breakdown: Hi Register Operations/Branch Exchange

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
-----															
0   1   0   0   0   1   Op   H1   H2   Rs/Hs     Rd/Hd															

### Details:

H1 = Hi operand flag 1

H2 = Hi operand flag 2

Op = Opcode

Op = 0 = 00, H1 = 0, H2 = 1 -> Opcode = ADD Rd, Hs

Op = 0 = 00, H1 = 1, H2 = 0 -> Opcode = ADD Hd, Rs

Op = 0 = 00, H1 = 1, H2 = 1 -> Opcode = ADD Hd, Hs

Op = 1 = 01, H1 = 0, H2 = 1 -> Opcode = CMP Rd, Hs

Op = 1 = 01, H1 = 1, H2 = 0 -> Opcode = CMP Hd, Rs

Op = 1 = 01, H1 = 1, H2 = 1 -> Opcode = CMP Hd, Hs

Op = 2 = 10, H1 = 0, H2 = 1 -> Opcode = MOV Rd, Hs

Op = 2 = 10, H1 = 1, H2 = 0 -> Opcode = MOV Hd, Rs

Op = 2 = 10, H1 = 1, H2 = 1 -> Opcode = MOV Hd, Hs

Op = 3 = 11, H1 = 0, H2 = 0 -> Opcode = BX Rs

Op = 3 = 11, H1 = 0, H2 = 1 -> Opcode = BX Hs

Rs/Hs = Source register, low

Rd/Hd = Destination register, low

Cond. Codes = Only the CMP opcode (Op = 01) sets the CPSR codes.

Notes = There are four sets of instructions in this group. The first three allow ADD, CMP, and MOV operations to be performed between Lo and Hi registers or a pair of Hi registers. The fourth instruction, BX, allows a branch to be performed which may also be

used to switch processor state from/to ARM Thumb-1. The action of H1 = 0, H2 = 0 for Op = 00 (ADD), Op = 01 (CMP), and Op = 10 (MOV) is undefined and as such should not be used.

**BX OpCode** = The BX opcode performs a branch to the start address specified in a Lo or Hi register. Bit 0 of the address determines the processor state on entry to the branch. If bit 0 is equal to zero the processor will be in ARM mode upon entering the branch. If bit 0 is equal to one then the processor will be in Thumb-1 state upon entering the branch. The action of H1 = 1 for this opcode is undefined, and should not be used.

**Using R15** = If R15 is used as an operand, the value will be the address of the instruction + 4 with bit 0 cleared. Executing a BX PC in Thumb-1 state from a non-word aligned, 4 byte boundary, address will result in unpredictable execution.

**Description:**

<b>Op</b>	<b>H1</b>	<b>H2</b>	<b>Thumb-1 Code</b>	<b>Action</b>
0 00	0	1	ADD Rd, Hs	Add a register in the range 8-15 to a register in the range 0-7.
0 00	1	0	ADD Hd, Rs	Add a register in the range 0-7 to a register in the range 8-15.
0 00	1	1	ADD Hd, Hs	Add two registers in the range 8-15.
1 01	0	1	CMP Rd, Hs	Compare a register in the range 0-7 with a register in the range 8-15. Set the condition code flags on result.
1 01	1	0	CMP Hd, Rs	Compare a register in the range 8-15 with a register in the range 0-7. Set the condition code flags on result.
1 01	1	1	CMP Hd, Hs	Compare two registers in the range 8-15. Set the condition code flags on the result.

2 10	0 1	1	MOV Rd, Hs	Move a value from a register in the range 8-15 to a register in the range 0-7.
2 10	1 0	0	MOV Hd, Rs	Move a value from a register in the range 0-7 to a register in the range 8-15.
2 10	1 0	1	MOV Hd, Hs	Move a value between two registers in the range 8-15.
3 11	0 1	0	BX Rs	Perform a branch (plus optional state change) to an address in a register in the range 0-7.
3 11	0 1	1	BX Hs	Perform a branch (plus optional state change) to an address in a registers in the range 8-15.

### Examples:

Endian	Code	Binary Rep	Hex Rep
Little	ADD R0, R8	01000000 01000100	0x40 44
Big	ADD R0, R8	01000100 01000000	0x44 40
Little	CMP R0, R8	01000000 01000101	0x40 45
Big	CMP R0, R8	01000101 01000000	0x45 40
Little	MOV R8, R9	11001000 01000110	0xC8 46
Big	MOV R8, R9	01000110 11001000	0x46 C8

### JSON Representation:

```
{
  "obj_name": "is_op_code",
  "op_code_name": "MOV",
  "index": 35,
  "arg_len": 2,
  "arg_separator": ",",
```

```

"bit_rep": {"bit_string": "0100011011", "bit_int": 283, "bit_len": 10},
"bit_series": {"bit_start": 6, "bit_stop": 15, "bit_len": 10},
"args": [
    {
        "obj_name": "is_op_code_arg",
        "arg_index": 0,
        "bit_index": 0,
        "is_entry_types": ["RegisterHi", "RegisterPc", "RegisterSp"],
        "is_arg_type": "Hd",
        "bit_series": {"bit_start": 0, "bit_stop": 2, "bit_len": 3}
    },
    {
        "obj_name": "is_op_code_arg",
        "arg_index": 1,
        "bit_index": 1,
        "is_entry_types": ["RegisterHi", "RegisterPc", "RegisterSp"],
        "is_arg_type": "Hs",
        "bit_series": {"bit_start": 3, "bit_stop": 5, "bit_len": 3}
    }
]
}

```

There isn't too much that is new in the previously listed JSON object for the `MOV` opcode. Notice that the `is_entry_types` allowed for each register argument are high register types, specifically providing for high registers PC and SP and their aliases. We'll see this in a few other opcode arguments so make sure you understand the distinction between high and low registers and special high registers that may have an alias.

## Format-6: PC Relative Load

### Bit Breakdown: PC Relative Load

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
-----															
0   1   0   0   1        Rd                                         Word8															

### Details:

Rd = Destination register, low

Word8 (#Imm) = The value specified by the Word8 argument is a full 10-bit address that must always be word aligned, i.e. with bits 0 and 1 set to zero. It's the assembler's responsibility to place the value `Word8 >> 2` in the Word8 opcode argument.

Cond. Codes = No condition codes are set by this opcode.  
 Notes = This instruction loads a word from an address specified as a 10-bit immediate offset from the PC. The value of the PC will be 4 bytes greater than the address of this instruction, but bit 1 of the PC is forced to 0 to ensure that it's word aligned.

#### Description:

Thumb-1 Code	Action
LDR Rd, [PC, #Imm]	Add unsigned offset (255 words, 1024 bytes) in #Imm to the current value of the PC. Load the word from the resulting address into Rd.

#### Examples:

Endian	Code	Binary Rep	Hex Rep
Little	LDR R0, [PC, #1020]	11111111 01001000	0xFF 48
Big	LDR R0, [PC, #1020]	01001000 11111111	0x48 FF

#### JSON Representation:

```
{
  "obj_name": "is_op_code",
  "op_code_name": "LDR",
  "index": 38,
  "arg_len": 3,
  "arg_separator": ",",
  "bit_rep": {"bit_string": "01001", "bit_int": 9, "bit_len": 5},
  "bit_series": {"bit_start": 11, "bit_stop": 15, "bit_len": 5},
  "args": [
    {
      "obj_name": "is_op_code_arg",
      "arg_index": 0,
      "bit_index": 1,
      "is_entry_types": ["RegisterLow"],
      "is_arg_type": "Rd",
      "bit_series": {"bit_start": 8, "bit_stop": 10, "bit_len": 3}
    },
    {
      "obj_name": "is_op_code_arg",
      "arg_index": 1,
      "bit_index": 1,
      "is_entry_types": ["RegisterLow"],
      "is_arg_type": "Rd",
      "bit_series": {"bit_start": 8, "bit_stop": 10, "bit_len": 3}
    }
  ]
}
```

```

    "arg_index": 1,
    "bit_index": 0,
    "is_entry_types": ["GroupStart"],
    "is_arg_type": "Group",
    "bit_series": {"bit_start": 0, "bit_stop": 7, "bit_len": 8},
    "sub_arg_separator": ",",
    "sub_args": [
        {
            "obj_name": "is_op_code_arg",
            "arg_index": 0,
            "bit_index": -1,
            "is_entry_types": ["RegisterPc"],
            "is_arg_type": "PC",
            "bit_series": {"bit_start": -1, "bit_stop": -1,
"bit_len": -1}
        },
        {
            "obj_name": "is_op_code_arg",
            "arg_index": 1,
            "bit_index": 0,
            "is_entry_types": ["Number", "LabelRef"],
            "is_arg_type": "Word8",
            "bit_series": {"bit_start": 0, "bit_stop": 7,
"bit_len": 8},
            "num_range": {"min_value": 0, "max_value": 1020,
"bit_len": 10, "twos_complement": false, "alignment": "WORD"},
            "bit_shift": {"shift_dir": "RIGHT", "shift_amount": 2}
        }
    ]
}
]
}

```

There are a few new topics for us to discuss with regard to the previously listed opcode. First off, we have the `GroupStart` argument data type. This is used to indicate a group of opcode arguments such as those used by the `LDR` instruction. Note that the group argument contains its own arguments. Notice that we do not include a `GroupEnd` argument data type. We could potentially have created a definition for one, but instead we let it be implied by the end of the sub-arguments list. In other words, the end of the argument when in a group or list indicated the end of the group or list.

The first group argument is the PC register. Take a moment to look at the `bit_index` of the PC register argument. Note that the argument index is set to zero instead of one. This is because the argument and bit indices get reset in a group or list argument object. But in this case the PC register has a

`bit_index` of negative one. This indicates that it isn't part of the final binary encoding. Notice in the bit break down there is no explicit listing for the PC register, it's used only as part of the opcode's syntax.

There's one more thing I'd like to discuss in this opcode JSON entry. The `Word8` argument data type has an inherent bit shift. This is the first time we've come across this concept in the instruction set description. The 2-bit shift allows us to use larger numbers, 10-bit, at the expense of not being able to use bits 0 and 1. Both of these bits should be set to zero because the data will be lost by the shift operation. Take a moment to note the structure of the bit shift sub-object in the `Word8` group's arguments.

## Format-7: Load/Store with Register Offset - STR, STRB, LDR, LDRB

### Bit Breakdown: Load/Store with Register Offset

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
-----															
0   1   0   1   L   B   0        Ro             Rb             Rd															

### Details:

L = Load/Store flag

L = 0 -> Store to memory

L = 1 -> Load from memory

B = Byte/Word flag

B = 0 -> Transfer word quantity

B = 1 -> Transfer byte quantity

Ro = Offset register, low

Rb = Base register

Rd = Source/Destination register

Cond. Codes = Condition codes are not set by this opcode group.

Notes = The instructions in this group are used to transfer byte or word values between registers and memory. Memory addresses are pre-indexed using an offset register in the range 0-7, low register.

### Description:

L	B	Thumb-1 Code	Action
0	0	STR Rd, [Rb, Ro]	Pre-indexed word store: Calculate the

			target address by adding the value in Rb and the value in Ro. Store the contents of Rd at the calculated address.
0	1	STRB Rd, [Rb, Ro]	Pre-indexed byte store: Calculate the target address by adding together the value in Rb and the value in Ro. Store the byte value in Rd at the calculated address.
1	0	LDR Rd, [Rb, Ro]	Pre-indexed word load: Calculate the source address by adding together the value in Rb and the value Ro. Load the contents of the address into Rd.
1	1	LDRB Rd, [Rb, Ro]	Pre-indexed byte load: Calculate the source address by adding together the value in Rb and the value in Ro. Load the byte value at the calculated address.

### Examples:

Endian	Code	Binary Rep	Hex Rep
Little	STR R0, [R1, R2]	10001000 01010000	0x88 50
Big	STR R0, [R1, R2]	01010000 10001000	0x50 88
Little	STRB R0, [R1, R2]	10001000 01010100	0x88 54
Big	STRB R0, [R1, R2]	01010100 10001000	0x54 88
Little	LDR R0, [R1, R2]	10001000 01011000	0x88 58
Big	LDR R0, [R1, R2]	01011000 10001000	0x58 88

### JSON Representation:

```
{
  "obj_name": "is_op_code",
  "op_code_name": "STR",
```

```

"index": 39,
"is_write_op_code": true,
"arg_len": 3,
"arg_separator": ",",
"bit_rep": {"bit_string": "0101000", "bit_int": 40, "bit_len": 7},
"bit_series": {"bit_start": 9, "bit_stop": 15, "bit_len": 7},
"args": [
  {
    "obj_name": "is_op_code_arg",
    "arg_index": 0,
    "bit_index": 0,
    "is_entry_types": ["RegisterLow"],
    "is_arg_type": "Rd",
    "bit_series": {"bit_start": 0, "bit_stop": 2, "bit_len": 3}
  },
  {
    "obj_name": "is_op_code_arg",
    "arg_index": 1,
    "bit_index": 1,
    "is_entry_types": ["GroupStart"],
    "is_arg_type": "Group",
    "bit_series": {"bit_start": 3, "bit_stop": 8, "bit_len": 6},
    "sub_arg_separator": ",",
    "sub_args": [
      {
        "obj_name": "is_op_code_arg",
        "arg_index": 0,
        "bit_index": 0,
        "is_entry_types": ["RegisterLow"],
        "is_arg_type": "Rb",
        "bit_series": {"bit_start": 3, "bit_stop": 5,
          "bit_len": 3}
      },
      {
        "obj_name": "is_op_code_arg",
        "arg_index": 1,
        "bit_index": 1,
        "is_entry_types": ["RegisterLow"],
        "is_arg_type": "Ro",
        "bit_series": {"bit_start": 6, "bit_stop": 8,
          "bit_len": 3}
      }
    ]
  }
]

```

```

    }
]
}

```

Aside from what we've already mentioned there isn't much new in the previous listing to talk about. Keep in mind that you can view examples of most, if not all, of the ARM Thumb-1 instructions in test program A's output files. Also, you can view all of the JSON objects for this instruction set in the `is_op_codes.json` file as mentioned previously.

## **Format-8: Load/Store Sign Extended Byte/Halfword - STRH, LDRH, LDSB, LDSH**

### **Bit Breakdown: Load/Store Sign Extended Byte/Halfword**

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
-----															
0   1   0   1   H   S   1        Ro             Rb             Rd															

### **Details:**

H = H flag  
 H = 1 → Used in the LDRH instruction  
 S = Sign extended flag  
 S = 0 → Operand not sign-extended  
 S = 1 → Operand sign-extended  
 Ro = Offset register, low  
 Rb = Base register  
 Rd = Source/Destination register  
 Cond. Codes = Condition codes are not set by this opcode group.  
 Notes = These instructions load optionally sign-extended bytes or halfwords, and store halfwords.

### **Description:**

S	H	Thumb-1 Code	Action
0	0	STRH Rd, [Rb, Ro]	Store halfword: Add Ro to base address in Rb. Store bit 0-15 of Rd at the resulting address.
0	1	LDRH Rd, [Rb, Ro]	Load halfword: Add Ro to the base address in Rb. Load bits 0-15

1	0	LDSB Rd, [Rb, Ro]	Load sign-extended byte: Add Ro to base address in Rb. Load bits 0-7 of Rd from the resulting address, and set bits 8-31 of Rd to bit 7.
1	1	LDSH Rd, [Rb, Ro]	Load sign-extended halfword: Add Ro to base address in Rb. Load bits 0-15 of Rd from the resulting address, and set bits 16-31 of Rd to bit 15.

### Examples:

Endian	Code	Binary Rep	Hex Rep
Little	STRH R0, [R1, R2]	10001000 01010010	0x88 52
Big	STRH R0, [R1, R2]	10001000 01010000	0x52 88
Little	LDRH R0, [R1, R2]	10001000 01011010	0x88 5A
Big	LDRH R0, [R1, R2]	01011010 10001000	0x5A 88
Little	LDSB R0, [R1, R2]	10001000 01010110	0x88 56
Big	LDSB R0, [R1, R2]	01010110 10001000	0x56 88

### JSON Representation:

```
{
  "obj_name": "is_op_code",
  "op_code_name": "STRH",
  "index": 43,
  "arg_len": 3,
  "arg_separator": ",",
  "bit_rep": {"bit_string": "0101001", "bit_int": 41, "bit_len": 7},
  "bit_series": {"bit_start": 9, "bit_stop": 15, "bit_len": 7},
  "args": [
    {
      "obj_name": "is_op_code_arg",
      "arg_index": 0,
      "bit_index": 0,
      "bit_start": 9,
      "bit_stop": 15,
      "bit_len": 7
    }
  ]
}
```

```

    "is_entry_types": ["RegisterLow"],
    "is_arg_type": "Rd",
    "bit_series": {"bit_start": 0, "bit_stop": 2, "bit_len": 3}
  },
  {
    "obj_name": "is_op_code_arg",
    "arg_index": 1,
    "bit_index": 1,
    "is_entry_types": ["GroupStart"],
    "is_arg_type": "Group",
    "bit_series": {"bit_start": 3, "bit_stop": 8, "bit_len": 6},
    "sub_arg_separator": ",",
    "sub_args": [
      {
        "obj_name": "is_op_code_arg",
        "arg_index": 0,
        "bit_index": 0,
        "is_entry_types": ["RegisterLow"],
        "is_arg_type": "Rb",
        "bit_series": {"bit_start": 3, "bit_stop": 5,
          "bit_len": 3}
      },
      {
        "obj_name": "is_op_code_arg",
        "arg_index": 1,
        "bit_index": 1,
        "is_entry_types": ["RegisterLow"],
        "is_arg_type": "Ro",
        "bit_series": {"bit_start": 6, "bit_stop": 8,
          "bit_len": 3}
      }
    ]
  }
}

```

The previous JSON object listing doesn't have anything we haven't seen before. As such, we won't spend time reviewing it. Instead, we'll move onto the next opcode group.

## Format-9: Load/Store with Immediate - STR, LDR, STRB, LDRB

### Bit Breakdown: Load/Store with Immediate

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
-----															
0   1   1   B   L                    Offset5                      Rb                      Rd															

### Details:

B = Word/Byte flag

B = 0 -> Transfer word quantity

B = 1 -> Transfer byte quantity

L = Store/Load flag

S = 0 -> Store to memory

S = 1 -> Read from memory

Offset5(#Imm) = Offset value, 7-bit value shifted right 2 bits to a 5-bit value, see notes below.

Rb = Base register

Rd = Source/Destination register

Cond. Codes = Condition codes are not set by this opcode group.

Notes = These instructions transfer byte or word values between registers and memory using an immediate 5 or 7-bit value. For word accesses, B=0, the value specified by #Imm is a full 7-bit address, but must be word-aligned, bits 1 and 0 set to zero, since the assembler places #Imm >> 2 in the Offset5 field.

### Description:

L	B	Thumb-1 Code	Action
0	0	STR Rd, [Rb, #Imm]	Calculate the target address by adding together the value in Rb and #Imm. Store the contents of Rd at the address.
1	0	LDR Rd, [Rb, #Imm]	Calculate the source address by adding together the value in Rb and #Imm. Load Rd from the address.
0	1	STRB Rd, [Rb, #Imm]	Calculate the target address by adding together the value in Rb and #Imm. Store the byte value in Rd at

the address.

1	1	LDRB Rd, [Rb, #Imm]	Calculate source address by adding together the value in Rb and #Imm. Load the byte value at the address into Rd.
---	---	---------------------	--

#### Examples:

Endian	Code	Binary Rep	Hex Rep
Little	STR R0, [R1, #0]	00001000 01100000	0x08 60
Big	STR R0, [R1, #0]	01100000 00001000	0x60 08
Little	STR R0, [R1, #124]	11001000 01100111	0xC8 67
Big	STR R0, [R1, #124]	01100111 11001000	0x67 C8
Little	LDR R0, [R1, #0]	00001000 01101000	0x08 68
Big	LDR R0, [R1, #0]	01101000 00001000	0x68 08

#### JSON Representation:

```
{  
    "obj_name": "is_op_code",  
    "op_code_name": "STR",  
    "index": 47,  
    "is_write_op_code": true,  
    "arg_len": 3,  
    "arg_separator": ",",  
    "bit_rep": {"bit_string": "01100", "bit_int": 12, "bit_len": 5},  
    "bit_series": {"bit_start": 11, "bit_stop": 15, "bit_len": 5},  
    "args": [  
        {  
            "obj_name": "is_op_code_arg",  
            "arg_index": 0,  
            "bit_index": 0,  
            "is_entry_types": ["RegisterLow"],  
            "bit_start": 11, "bit_stop": 15, "bit_len": 5  
        }  
    ]  
}
```

```

    "is_arg_type": "Rd",
    "bit_series": {"bit_start": 0, "bit_stop": 2, "bit_len": 3}
},
{
    "obj_name": "is_op_code_arg",
    "arg_index": 1,
    "bit_index": 1,
    "is_entry_types": ["GroupStart"],
    "is_arg_type": "Group",
    "bit_series": {"bit_start": 3, "bit_stop": 10, "bit_len": 8},
    "sub_arg_separator": ",",
    "sub_args": [
        {
            "obj_name": "is_op_code_arg",
            "arg_index": 0,
            "bit_index": 0,
            "is_entry_types": ["RegisterLow"],
            "is_arg_type": "Rb",
            "bit_series": {"bit_start": 3, "bit_stop": 5,
"bit_len": 3}
        },
        {
            "obj_name": "is_op_code_arg",
            "arg_index": 1,
            "bit_index": 1,
            "is_entry_types": ["Number", "LabelRef"],
            "is_arg_type": "Offset5",
            "bit_series": {"bit_start": 6, "bit_stop": 10,
"bit_len": 5},
            "num_range": {"min_value": 0, "max_value": 124,
"bit_len": 7, "twos_complement": false, "alignment": "WORD"},
            "bit_shift": {"shift_dir": "RIGHT", "shift_amount": 2}
        }
    ]
}
]
}

```

I won't go into any detail reviewing the previously listed opcode group. The JSON object is very similar to the **Format-6: PC Relative Load** opcode group we reviewed earlier. Take a moment to look over this JSON object and make sure you understand its structure before moving on to the next section.

## Chapter Conclusion

In this chapter we covered a lot of material. We got into the nitty gritty details of the ARM Thumb-1 instruction set including discussing some important details like listing the available registers, their bit depth, and the different modes the CPU can run in. Let's take a look at the topics we covered in this chapter.

1. **The ARM Thumb-1 Instruction Set:** In this section we started the process of learning the ARM Thumb-1 instruction set. We covered some information regarding the available registers. We also reviewed the CPSR register and noted the meaning of that register's bits including the different supported processor modes.
2. **The Thumb-1 Opcode Entry's Structure:** In this section we took a look at the structure of an opcode's JSON data entry. We also started our review of the Thumb-1 instruction set with the "Format-1: Move Shifted Register - LSL, LSR, ASR" group of opcodes. We introduced you to the structure of the JSON data entry and described some details of how the data entry is used.
3. **Thumb-1 Opcodes Part 1:** We continued our in-depth review of the ARM Thumb-1 instruction set. Included in the Part 1 review are the following opcode groups. Each group member, opcode, has a similar binary structure.
  - **Format-2:** Add/Subtract – ADD, SUB
  - **Format-3:** Move/Compare/Add/Subtract Immediate – MOV, CMP, ADD, SUB
  - **Format-4:** ALU Operations

- **Format-5:** Hi Register Operations/Branch Exchange – ADD, CMP, MOV, BX
- **Format-6:** PC Relative Load
- **Format-7:** Load/Store with Register Offset – STR, STRB, LDR, LDRB
- **Format-8:** Load/Store Sign Extended Byte/Halfword – STRH, LDRH, LDSB, LDSH
- **Format-9:** Load/Store with Immediate – STR, LDR, STRB, LDRB

We jumped right into our ARM Thumb-1 instruction set review and took a look at nine different opcode groups. We've reviewed a lot of details about each instruction group. Don't be worried if it doesn't all make perfect sense to you. As you get more experience working with the opcodes, while building the assembler as well as while writing test programs for the assembler, you'll gain a deeper understanding of them.

## Chapter 4: ARM Thumb-1 Opcodes Part 2

Welcome to part 2 of the ARM Thumb-1 instruction set review. In part 1 we took a look at nine different opcode groups. An opcode group shares a similar encoding and argument structure. There are a few more for us to review before we can move on and focus on building an assembler that supports the ARM Thumb-1 instruction set. I want to mention that information on each opcode group is gathered from the **ARM7TDMI** data sheet, version **DDI-0029E**. I make no claim on text from this document and I've made adjustments and added information where it applies within the context of this project.

Again, you can find a copy of the documentation online by searching for “ARM7TDMI DDI-0029E” in your favorite search engine. I’ve also included a copy of the documentation used with the GenAsm project. It is located in the NetBeans project’s local directory at this location, “.`\storage\documentation\ARM7TDMI_DDI-0029E.pdf`”.

I should mention that I specifically targeted the Thumb-1 instruction set due to its concise yet functional nature. As you’ll see a little later on in this text the instruction set is robust enough to power a series of test programs that run on a GameBoy Advance emulator and display text on the screen. We’ve seen a few different opcodes and opcode arguments in part 1. We’ll look at

the remaining opcode groups and their associated arguments here. Let's get to it, shall we?

## Format-10: Load/Store Halfword - STRH, LDRH

### Bit Breakdown: Load/Store Halfword

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
-----															
1   0   0   0   L                    Offset5                      Rb                      Rd															

### Details:

L = Load/Store bit

L = 0 → Opcode = STRH

L = 1 → Opcode = LDRH

Offset5 = 5-bit immediate value

Rb = Source register, low

Rd = Destination register, low

Cond. Codes = Condition codes are not set by this opcode group.

Notes = These instructions transfer halfword values between a Low register and a memory address. Addresses are pre-indexes, using a 6-bit immediate value. The Offset5, #Imm, value is a full 6-bit address but must be halfword-aligned (i.e. with bit 0 set to 0) since the assembler places #Imm >> 1 in the Offset5 field.

### Description:

L	Thumb-1 Code	Action
0	STRH Rd, [Rb, #Imm]	Add #Imm to base addresses in Rb and store bits 0-15 of Rd at the resulting address.
1	LDRH Rd, [Rb, #Imm]	Add #Imm to base address in Rb. Load bits 0-15 from the resulting address into Rd and set bits 16-31 zero.

### Examples:

Endian	Code	Binary Rep	Hex Rep
Little	STRH R0, [R1, #0]	00001000 10000000	0x08 80

Big	STRH R0, [R1, #0]	10000000 00001000	0x80 08
Little	STRH R0, [R1, #62]	11001000 10000111	0xC8 87
Big	STRH R0, [R1, #62]	10000111 11001000	0x87 C8
Little	LDRH R0, [R1, #0]	00001000 10001000	0x08 88
Big	LDRH R0, [R1, #0]	10001000 00001000	0x88 08

#### JSON Representation:

```
{
  "obj_name": "is_op_code",
  "op_code_name": "STRH",
  "index": 51,
  "arg_len": 3,
  "arg_separator": ",",
  "bit_rep": {"bit_string": "10000", "bit_int": 16, "bit_len": 5},
  "bit_series": {"bit_start": 11, "bit_stop": 15, "bit_len": 5},
  "args": [
    {
      "obj_name": "is_op_code_arg",
      "arg_index": 0,
      "bit_index": 0,
      "is_entry_types": ["RegisterLow"],
      "is_arg_type": "Rd",
      "bit_series": {"bit_start": 0, "bit_stop": 2, "bit_len": 3}
    },
    {
      "obj_name": "is_op_code_arg",
      "arg_index": 1,
      "bit_index": 1,
      "is_entry_types": ["GroupStart"],
      "is_arg_type": "Group",
      "bit_series": {"bit_start": 3, "bit_stop": 10, "bit_len": 8},
      "sub_arg_separator": ",",
      "sub_args": [
        {
          "obj_name": "is_op_code_arg",
          "arg_index": 0,

```

```

        "bit_index": 0,
        "is_entry_types": ["RegisterLow"],
        "is_arg_type": "Rb",
        "bit_series": {"bit_start": 3, "bit_stop": 5,
"bit_len": 3}
    },
    {
        "obj_name": "is_op_code_arg",
        "arg_index": 1,
        "bit_index": 1,
        "is_entry_types": ["Number", "LabelRef"],
        "is_arg_type": "Offset5",
        "bit_series": {"bit_start": 6, "bit_stop": 10,
"bit_len": 5},
        "num_range": {"min_value": 0, "max_value": 62,
"bit_len": 6, "twos_complement": false, "alignment": "HALFWORD"},
        "bit_shift": {"shift_dir": "RIGHT", "shift_amount": 1}
    }
]
}
}

```

The JSON object for this group of opcodes should be familiar. There isn't really anything new to cover though I should invite you to think about the bit shift used on the 'Offset5, #Imm' opcode argument. Why do you think the shift is only one bit? If you thought about half-word alignment you might be right. A half-word is 2 bytes and in order to have the #Imm value be half-word aligned it has to be an even number because bit 0 is forced to 0 during the 1-bit shift. For this reason odd numbers are dropped because the one's bit can only be zero.

## Format-11: SP-Relative Load/Store - STR, LDR

### Bit Breakdown: SP-Relative Load/Store

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
-----															
1   0   0   1   L        Rd                         Word8															

### Details:

L = Load/Store bit

L = 0 -> Opcode = STR  
 L = 1 -> Opcode = LDR  
 Word8 = 8-bit immediate value  
 Rd = Destination register, low  
 Cond. Codes = Condition codes are not set by this opcode group.  
 Notes = The instructions in the group perform an SP - relative load or store. The Thumb-1 assembler syntax is shown in the following table. The offset supplied in the #Imm argument is a full 10-bit, but must always be word-aligned (i.e. bits 1 and 0 set to 0), since the assembler places #Imm >> 2 in the Word8 field.

#### Description:

L	Thumb-1 Code	Action
0	STR Rd, [SP, #Imm]	Add unsigned offset (255 words, 1020 bytes) in #Imm to the current value of the SP. Store the contents of Rd at the resulting address.
1	LDR Rd, [SP, #Imm]	Add unsigned offset (255 words, 1020 bytes) in #Imm to the current value of the SP. Load the word from the resulting address into Rd.

#### Examples:

Endian	Code	Binary Rep	Hex Rep
Little	STR R0, [SP, #0]	00000000 10010000	0x00 90
Big	STR R0, [SP, #0]	10010000 00000000	0x90 00
Little	STR R0, [SP, #1020]	11111111 10010000	0xFF 90
Big	STR R0, [SP, #1020]	10010000 11111111	0x90 FF
Little	LDR R0, [SP, #0]	00000000 10011000	0x00 98
Big	LDR R0, [SP, #0]	10011000 00000000	0x98 00

#### JSON Representation:

```

{
  "obj_name": "is_op_code",
  "op_code_name": "LDR",
  "index": 54,
  "arg_len": 3,
  "arg_separator": ",",
  "bit_rep": {"bit_string": "10011", "bit_int": 19, "bit_len": 5},
  "bit_series": {"bit_start": 11, "bit_stop": 15, "bit_len": 5},
  "args": [
    {
      "obj_name": "is_op_code_arg",
      "arg_index": 0,
      "bit_index": 1,
      "is_entry_types": ["RegisterLow"],
      "is_arg_type": "Rd",
      "bit_series": {"bit_start": 8, "bit_stop": 10, "bit_len": 3}
    },
    {
      "obj_name": "is_op_code_arg",
      "arg_index": 1,
      "bit_index": 0,
      "is_entry_types": ["GroupStart"],
      "is_arg_type": "Group",
      "bit_series": {"bit_start": 0, "bit_stop": 7, "bit_len": 8},
      "sub_arg_separator": ",",
      "sub_args": [
        {
          "obj_name": "is_op_code_arg",
          "arg_index": 0,
          "bit_index": -1,
          "is_entry_types": ["RegisterSp"],
          "is_arg_type": "Rs",
          "bit_series": {"bit_start": -1, "bit_stop": -1,
            "bit_len": -1}
        },
        {
          "obj_name": "is_op_code_arg",
          "arg_index": 1,
          "bit_index": 0,

```

```

        "is_entry_types": ["Number", "LabelRef"],
        "is_arg_type": "Word8",
        "bit_series": {"bit_start": 0, "bit_stop": 7,
"bit_len": 8},
            "num_range": {"min_value": 0, "max_value": 1020,
"bit_len": 10, "twos_complement": false, "alignment": "WORD"},
                "bit_shift": {"shift_dir": "RIGHT", "shift_amount": 2}
            }
        ]
    }
]
}

```

It doesn't come up too often, so I'll take a moment to discuss it here. Notice in the previously listed JSON object that the `Rs` opcode argument is not part of the binary encoding for the given opcode. This can be seen in the `bit_series` sub-object of the `Rs` opcode argument as well as in the `bit_index` attribute for that argument. The use of the register is part of the instruction's assembly syntax but not actually part of the resulting binary representation. In the next section we'll take a look at yet another version of the `ADD` instruction.

## Format-12: Load Address - ADD

### Bit Breakdown: Load Address

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
-----															
	1		0		1		0		S		Rd		Word8		

### Details:

S = Source bit

S = 0 → Source = PC

S = 1 → Source = SP

Word8 = 8-bit unsigned constant

Rd = Destination register, low

Cond. Codes = Condition codes are not set by this opcode group.

Notes = These instructions calculate an address by adding a 10-bit constant to either the PC or the SP register, and load the resulting address into a register. The value specified by the Word8, #Imm, opcode argument is a full 10-bit value, but it must be work-

aligned (I.e. with bits 1 and 0 set to 0) since the assembler places `#Imm >> 2` in field Word8. Where the PC is used as the source register (`S = 0`), bit 1 of the PC is always read as 0. The value of the PC will be 4 bytes greater than the address of the instruction before bit 1 is forced to 0.

#### Description:

<b>S</b>	<b>Thumb-1 Code</b>	<b>Action</b>
0	<code>ADD Rd, PC, #Imm</code>	Add <code>#Imm</code> to the current value of the program counter, PC, and load the result into <code>Rd</code> .
1	<code>ADD Rd, SP, #Imm</code>	Add <code>#Imm</code> to the current value of the stack pointer, SP, and load the result into <code>Rd</code> .

#### Examples:

<b>Endian</b>	<b>Code</b>	<b>Binary Rep</b>	<b>Hex Rep</b>
Little	<code>ADD R0, PC, #0</code>	00000000 10100000	0x00 A0
Big	<code>ADD R0, PC, #0</code>	10100000 00000000	0xA0 00
Little	<code>ADD R0, PC, #1020</code>	11111111 10100000	0xFF A0
Big	<code>ADD R0, PC, #1020</code>	10100000 11111111	0xA0 FF
Little	<code>ADD R0, SP, #0</code>	00000000 10101000	0x00 A8
Big	<code>ADD R0, SP, #0</code>	10101000 00000000	0xA8 00

#### JSON Representation:

```
{
  "obj_name": "is_op_code",
  "op_code_name": "ADD",
  "index": 56,
  "arg_len": 3,
  "arg_separator": ",",
}
```

```

"bit_rep": {"bit_string": "10101", "bit_int": 21, "bit_len": 5},
"bit_series": {"bit_start": 11, "bit_stop": 15, "bit_len": 5},
"args": [
  {
    "obj_name": "is_op_code_arg",
    "arg_index": 0,
    "bit_index": 1,
    "is_entry_types": ["RegisterLow"],
    "is_arg_type": "Rd",
    "bit_series": {"bit_start": 8, "bit_stop": 10, "bit_len": 3}
  },
  {
    "obj_name": "is_op_code_arg",
    "arg_index": 1,
    "bit_index": -1,
    "is_entry_types": ["RegisterSp"],
    "is_arg_type": "Rs",
    "bit_series": {"bit_start": -1, "bit_stop": -1, "bit_len": -1}
  },
  {
    "obj_name": "is_op_code_arg",
    "arg_index": 2,
    "bit_index": 0,
    "is_entry_types": ["Number", "LabelRef"],
    "is_arg_type": "Word8",
    "bit_series": {"bit_start": 0, "bit_stop": 7, "bit_len": 8},
    "num_range": {"min_value": 0, "max_value": 1020, "bit_len": 10},
    "twos_complement": false, "alignment": "WORD"},
    "bit_shift": {"shift_dir": "RIGHT", "shift_amount": 2}
  }
]
}

```

Again, there's not much new to discuss about this opcode groups' JSON object. Take a close look at the opcode's arguments. This is another example of an instruction argument that is there to differentiate the argument signature and is not used as part of the actual instruction's binary representation.

# Format-13: Add Offset to Stack Pointer - ADD

## Bit Breakdown: Add Offset to Stack Pointer

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
-----															
1   0   1   1   0   0   0   0   0   S	SWord7														

### Details:

S = Sign bit  
S = 0 → Offset is positive  
S = 1 → Offset is negative  
SWord7 = 7-bit immediate value  
Cond. Codes = Condition codes are not set by this opcode group.  
Notes = The instruction adds a 9-bit signed constant to the stack pointer. The offset specified, SWord7 (#Imm) can be up to -/+ 508, but must be word-aligned (i.e. with bits 1 and 0 set to 0) since the assembler converts the #Imm value to an 8-bit signed + magnitude number before placing it in argument SWord7.

### Description:

S	Thumb-1 Code	Action
0	ADD SP, #Imm	Add the #Imm value to the stack pointer, SP.
1	ADD SP, #-Imm	Add the #-Imm value to the stack pointer, SP.

### Examples:

Endian	Code	Binary Rep	Hex Rep
Little	ADD SP, #0	00000000 10110000	0x00 B0
Big	ADD SP, #0	10110000 00000000	0xB0 00
Little	ADD SP, #508	01111111 10110000	0x7F B0
Big	ADD SP, #508	10110000 01111111	0xB0 7F
Little	ADD SP, #-2	** N/A	N/A

Big	ADD SP, #-2	** N/A	N/A
-----	-------------	--------	-----

\*\* The ADD, SP -#Imm has not been implemented because I could not get it working correctly in the VASM assembler and there are other ADD instructions that can be used to overcome this problem.

\*\*

#### JSON Representation:

```
{  
    "obj_name": "is_op_code",  
    "op_code_name": "ADD",  
    "index": 57,  
    "arg_len": 2,  
    "arg_separator": ",",  
    "bit_rep": {"bit_string": "101100000", "bit_int": 352, "bit_len": 9},  
    "bit_series": {"bit_start": 7, "bit_stop": 15, "bit_len": 9},  
    "args": [  
        {  
            "obj_name": "is_op_code_arg",  
            "arg_index": 0,  
            "bit_index": -1,  
            "is_entry_types": ["RegisterSp"],  
            "is_arg_type": "Rs",  
            "bit_series": {"bit_start": -1, "bit_stop": -1, "bit_len": -1}  
        },  
        {  
            "obj_name": "is_op_code_arg",  
            "arg_index": 1,  
            "bit_index": 0,  
            "is_entry_types": ["Number", "LabelRef"],  
            "is_arg_type": "SWord7",  
            "bit_series": {"bit_start": 0, "bit_stop": 6, "bit_len": 7},  
            "num_range": {"min_value": -508, "max_value": 508, "bit_len": 9, "twos_complement": false, "alignment": "WORD"},  
            "bit_shift": {"shift_dir": "RIGHT", "shift_amount": 2}  
        }  
    ]  
}
```

This opcode group actually has a subtle but unique attribute to it that makes it a little bit more complicated to work with and a bit different than any of the other opcode groups we've reviewed. In this case the binary encoding is altered based on an attribute of one of the instruction's arguments. In this case if the value of  $\#Imm$  is less than zero.

In order to handle this peculiarity, there are a few others we'll encounter as well, we had to add a special check in the code to adjust the L bit after the argument value has been determined. This is the first time we don't know how to handle a bit flag until after the arguments are processed. Usually, the syntax of the assembly language determines the state of an opcode's bit flags. Take special note of this distinction.

## **Format-14: Push/Pop Registers - PUSH, POP**

### **Bit Breakdown: Push/Pop Registers**

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
<hr/>															
	1		0		1		1		L		1		0		R

Rlist

### **Details:**

L = Load store bit

L = 0 → Store to memory

L = 1 → Load from memory

R = PC/LR bit

R = 0 → Do not store LR/load PC

R = 1 → Store LR/Load PC

Rlist = Register list

Cond. Codes = Condition codes are not set by this opcode group.

Notes = The instructions in this group allow registers 0-7 and optionally LR to be pushed onto the stack, and registers 0-7 and optionally PC to be popped off the stack. The stack is always assumed to be Full Descending.

### **Description:**

S	R	Thumb-1 Code	Action
0	0	PUSH {Rlist}	Push the registers specified by the Rlist onto the stack. Update the stack

			pointer.
0	1	PUSH {Rlist, LR}	Push the Link Register and the registers specified by Rlist (if any) onto the stack. Update the stack pointer.
1	0	POP {Rlist}	Pop values off the stack into the registers specified by Rlist. Update the stack pointer.
1	1	POP {Rlist, PC}	Pop values off the stack and load them into the registers specified by Rlist. Pop the PC off the stack. Update the stack pointer.

#### Examples:

Endian	Code	Binary Rep	Hex Rep
Little	PUSH {R0, R1}	00000011 10110100	0x03 B4
Big	PUSH {R0, R1}	10110100 00000011	0xB4 03
Little	PUSH {R0, R1, LR}	00000011 10110101	0x03 B5
Big	PUSH {R0, R1, LR}	10110101 00000011	0xB5 03
Little	POP {R0, R1}	00000011 10111100	0x03 BD
Big	POP {R0, R1}	10111100 00000011	0xBD 03

#### JSON Representation:

```
{
  "obj_name": "is_op_code",
  "op_code_name": "PUSH",
  "index": 59,
  "arg_len": 1,
  "arg_separator": ",",
  "bit_rep": {"bit_string": "10110100", "bit_int": 180, "bit_len": 8},
```

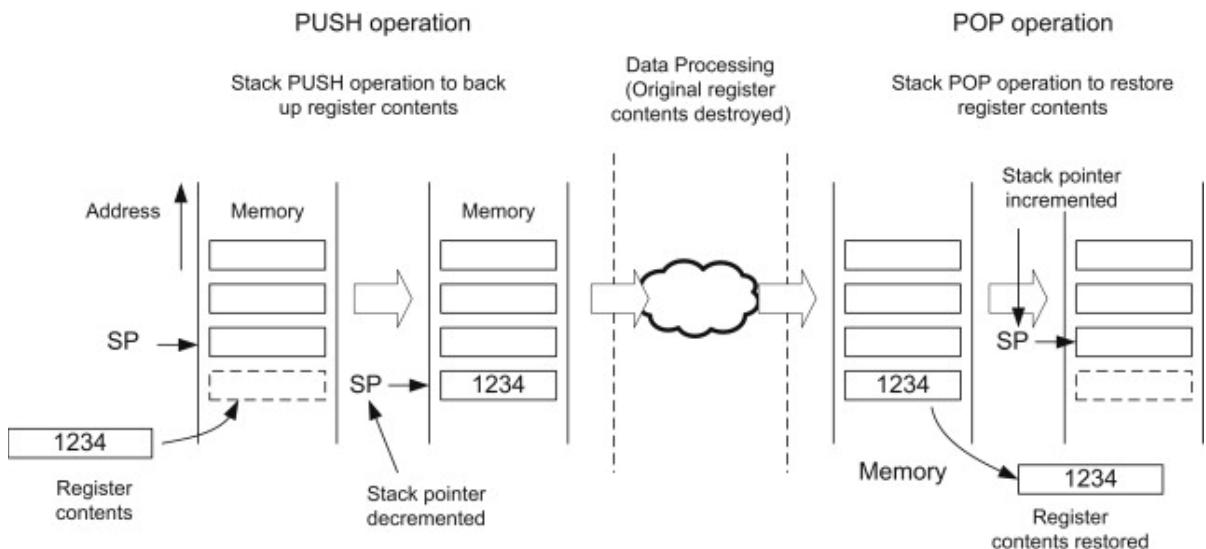
```

"bit_series": {"bit_start": 8, "bit_stop": 15, "bit_len": 8},
"args": [
    {
        "obj_name": "is_op_code_arg",
        "arg_index": 0,
        "bit_index": 0,
        "is_entry_types": ["ListStart"],
        "is_arg_type": "List",
        "bit_series": {"bit_start": 0, "bit_stop": 7, "bit_len": 8},
        "list_separator": ",",
        "sub_args": [
            {
                "obj_name": "is_op_code_arg",
                "arg_index": 0,
                "bit_index": 0,
                "is_entry_types": ["RegisterRangeLow"],
                "is_arg_type": "Rx",
                "bit_series": {"bit_start": 0, "bit_stop": 7,
"bit_len": 8}
            }
        ]
    }
]
}

```

A quick note about the stack in assembly programming. The role of stack memory includes storage of temporary data when handling function calls (normal stack PUSH and POP operations), storage for local variables, passing of parameters in function calls, saving of registers during exception sequences, etc. For this project we'll be targeting the stack operation is based on a “full-descending” stack model. This means the stack pointer always points to the last filled data in the stack memory, and the stack pointer pre-decrements for each new data store (PUSH).

*Image 4-1: An image depicting full-descending stack operations*



An image from Rutgers University science direct depicting the push and pop operations on a full-descending stack.

The push/pop opcode group takes a register list argument type. The GenAsm assembler, as you'll soon see, supports two ways to specify a register list. You can use a register range, R0–R4, or an explicit list of registers like, R0, R2, R4, and so on. Each register has a corresponding bit, using bits 0-7 in the instruction's bit break down. When a register is present in the list that bit is set to 1 otherwise it is set to 0. This is the first time we've encountered the list argument type.

Take a moment to look over this JSON object carefully before moving on. In the next section we'll take a look at the second instruction permutation of this opcode group. This is a peculiarity of this opcode group. Usually, an opcode group has only one instruction permutation. Let's take a look at the second **PUSH** instruction in the group.

#### JSON Representation:

```
{
  "obj_name": "is_op_code",
  "op_code_name": "PUSH",
  "index": 60,
```

```

"arg_len": 1,
"arg_separator": ",",
"bit_rep": {"bit_string": "10110101", "bit_int": 181, "bit_len": 8},
"bit_series": {"bit_start": 8, "bit_stop": 15, "bit_len": 8},
"args": [
{
    "obj_name": "is_op_code_arg",
    "arg_index": 0,
    "bit_index": 0,
    "is_entry_types": ["ListStart"],
    "is_arg_type": "List",
    "bit_series": {"bit_start": 0, "bit_stop": 7, "bit_len": 8},
    "list_separator": ",",
    "sub_args": [
        {
            "obj_name": "is_op_code_arg",
            "arg_index": 0,
            "bit_index": 0,
            "is_entry_types": ["RegisterRangeLow"],
            "is_arg_type": "Rx",
            "bit_series": {"bit_start": 0, "bit_stop": 7,
"bit_len": 8}
        },
        {
            "obj_name": "is_op_code_arg",
            "arg_index": 1,
            "bit_index": -1,
            "is_entry_types": ["RegisterLr"],
            "is_arg_type": "LR",
            "bit_series": {"bit_start": -1, "bit_stop": -1,
"bit_len": -1}
        }
    ]
}
]
}

```

The main difference between this data entry and the previously listed one is that there is a symbolic argument. We've seen this once or twice

before. This occurs when the assembly source code includes an argument which is not part of the instructions binary encoding. In this way the argument is used more as a way to differentiate the two instructions from each other when expressed in assembly source code. Notice that a symbolic argument is designated by the -1 value used in the `bit_index` and `bit_series` attributes of the opcode argument.

## Format-15: Multiple Load/Store - STMIA, LDMIA

### Bit Breakdown: Multiple Load/Store

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
<hr/>															
	1		1		0		0		L		Rb		Rlist		

### Details:

L = Load store bit  
 L = 0 → Store to memory  
 L = 1 → Load from memory  
 Rlist = Register list  
 Rb = Base register  
 Cond. Codes = Condition codes are not set by this opcode group.  
 Notes = These instructions allow multiple loading and storing of Low registers.

### Description:

L	Thumb-1 Code	Action
0	STMIA Rb!, {Rlist}	Store the registers specified by Rlist, starting at the base address in Rb. Write back the new base address.
1	LDMIA Rb!, {Rlist}	Load the registers specified by Rlist, starting at the base address in Rb. Write back the new base address.

### Examples:

Endian	Code	Binary Rep	Hex Rep
Little	STMIA R0!, {R1, R2}	00000110	0x06 C0

		11000000	
Big	STMIA R0!, {R1, R2}	11000000 00000110	0xC0 06
Little	LDMIA R0!, {R1, R2}	00000110 11001000	0x06 C8
Big	LDMIA R0!, {R1, R2}	11001000 00000110	0xC8 06

#### JSON Representation:

```
{
  "obj_name": "is_op_code",
  "op_code_name": "STMIA",
  "index": 63,
  "arg_len": 2,
  "arg_separator": ",",
  "bit_rep": {"bit_string": "11000", "bit_int": 24, "bit_len": 5},
  "bit_series": {"bit_start": 11, "bit_stop": 15, "bit_len": 5},
  "args": [
    {
      "obj_name": "is_op_code_arg",
      "arg_index": 0,
      "bit_index": 1,
      "is_entry_types": ["RegisterWb"],
      "is_arg_type": "Rwb",
      "bit_series": {"bit_start": 8, "bit_stop": 10, "bit_len": 3}
    },
    {
      "obj_name": "is_op_code_arg",
      "arg_index": 1,
      "bit_index": 0,
      "is_entry_types": ["ListStart"],
      "is_arg_type": "List",
      "bit_series": {"bit_start": 0, "bit_stop": 7, "bit_len": 8},
      "list_separator": ",",
      "sub_args": [
        {
          "obj_name": "is_op_code_arg",
          "arg_index": 0,
          "bit_index": 0,
          "is_entry_types": ["ListElement"],
          "is_arg_type": "ListElement"
        }
      ]
    }
  ]
}
```

```

        "obj_name": "is_op_code_arg",
        "arg_index": 0,
        "bit_index": 0,
        "is_entry_types": ["RegisterRangeLow"],
        "is_arg_type": "Rx",
        "bit_series": {"bit_start": 0, "bit_stop": 7,
"bit_len": 8}
    }
]
}
]
}

```

The previously listed JSON object has one attribute that we'll briefly discuss here. The first argument is a register write back data type, `Rwb`. This data type is used to indicate which register to write back to, in this case, capturing the new base address. It is expressed syntactically as a register label followed by an exclamation point, `R0!`, `R1!`, etc.

## **Format-16: Conditional Branch - BEQ, BNE, BCS, BCC, BMI, BPL, BVS, BVC, BHI, BLS, BGE, BLT, BGT, BLE**

### **Bit Breakdown: Multiple Load/Store**

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
<hr/>															
	1		1		0		1		Cond			SOffset8			

### **Details:**

Cond = Condition code

SOffset8 = 8-bit Signed immediate

Cond. Codes = Condition codes are not set by this opcode group.

Notes = The instructions in this group all perform a conditional Branch depending on the state of the CPSR condition codes. The branch offset must take account of the prefetch operation, which causes the PC to be 1 word (4 bytes) ahead of the current instruction.

= While label specifies a full 9-bit two's complement address, this must always be halfword-aligned (i.e. with bit 0 set to 0) since the assembler actually places label >> 1 in field SOffset8.

= Condition code 1110 is undefined and should not be used. Condition code 1111 creates the SWI instruction.

**Description:**

<b>Cond</b>	<b>Thumb-1 Code</b>	<b>Action</b>
0000	BEQ labelRef	Branch if Z is set, equal
0001	BNE labelRef	Branch if z is clear, not equal
0010	BCS labelRef	Branch if C set, unsigned higher or same
0011	BCC labelRef	Branch if C clear, unsigned lower
0100	BMI labelRef	Branch if N set, negative
0101	BPL labelRef	Branch if N clear, positive or zero
0110	BVS labelRef	Branch if V set, overflow
0111	BVC labelRef	Branch if V clear, no overflow
1000	BHI labelRef	Branch if C set and Z clear, unsigned higher
1001	BLS labelRef	Branch if C clear or Z set, unsigned lower or same
1010	BGE LabelRef	Branch if N set and V set, or N clear and V clear, greater or equal
1011	BLT labelRef	Branch if N set and V clear, or N clear and V set, less than
1100	BGT labelRef	Branch if Z clear, and either N set and V set or N clear and V clear, greater than
1101	BLE labelRef	Branch if Z set, of N set and V clear, or N clear and V set, less than or equal

### Examples:

Endian	Code	Binary Rep	Hex Rep
Little	BEQ `label_	10101010 11010000	0xAA D0
Big	BEQ `label_	11010000 10101010	0xD0 AA
Little	BNE `label_	10101001 11010001	0xA9 D1
Big	BNE `label_	11010001 10101001	0xD1 A9
Little	BCS `label_	10101000 11010010	0xA8 D2
Big	BCS `label_	11010010 10101000	0xD2 A8

### JSON Representation:

```
{  
    "obj_name": "is_op_code",  
    "op_code_name": "BEQ",  
    "index": 65,  
    "arg_len": 1,  
    "arg_separator": ",",  
    "bit_rep": {"bit_string": "11010000", "bit_int": 208, "bit_len": 8},  
    "bit_series": {"bit_start": 8, "bit_stop": 15, "bit_len": 8},  
    "args": [  
        {  
            "obj_name": "is_op_code_arg",  
            "arg_index": 0,  
            "bit_index": 0,  
            "is_entry_types": ["Number", "LabelRef"],  
            "is_arg_type": "Label",  
            "bit_series": {"bit_start": 0, "bit_stop": 7, "bit_len": 8},  
        }  
    ]  
}
```

```

        "num_range": {"min_value": 0, "max_value": 255, "bit_len": 9,
"twos_complement": true, "handle_prefetch": true, "use_halfword_offset": true, "alignment": "HALFWORD"},

        "bit_shift": {"shift_dir": "RIGHT", "shift_amount": 1}
    }
]
}

```

In the JSON object for this opcode group, we have an argument with an `is_arg_type` of `Label`. This argument type can be satisfied with a numeric value but is intended to be used with assembly code labels managed by the assembler. The GenAsm assembler supports a few different ways to reference labels. We'll revisit this in a latter chapter when we start reviewing some code.

## Format-17: Software Interrupt - SWI

### Bit Breakdown: Software Interrupt

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00	
-----																
	1		1		0		1		1		1		1		Value8	

### Details:

Value8 = 8-bit Signed immediate  
Cond. Codes = Condition codes are not set by this opcode group.  
Notes = The SWI instruction performs a software interrupt. On taking the SWI, the processor switches into ARM state and enters Supervisor (SVC) mode.

### Description:

#### Thumb-1 Code      Action

SWI Value8	Perform Software Interrupt: Move the address of the next instruction in the LR, move CPSR to SPSR, load the SWI vector address (0x8) into the PC register. Switch to ARM state and enter SVC mode.
------------	---

### **Examples:**

<b>Endian</b>	<b>Code</b>	<b>Binary Rep</b>	<b>Hex Rep</b>
Little	SWI #0	00000000 11011111	0x00 DF
Big	SWI #0	11011111 00000000	0xDF 00
Little	SWI #255	11111111 11011111	0xFF DF
Big	SWI #255	11011111 11111111	0xDF FF

### **JSON Representation:**

```
{
  "obj_name": "is_op_code",
  "op_code_name": "SWI",
  "index": 79,
  "arg_len": 1,
  "arg_separator": ",",
  "bit_rep": {"bit_string": "11011111", "bit_int": 223, "bit_len": 8},
  "bit_series": {"bit_start": 8, "bit_stop": 15, "bit_len": 8},
  "args": [
    {
      "obj_name": "is_op_code_arg",
      "arg_index": 0,
      "bit_index": 0,
      "is_entry_types": ["Number", "LabelRef"],
      "is_arg_type": "Value8",
      "bit_series": {"bit_start": 0, "bit_stop": 7, "bit_len": 8},
      "num_range": {"min_value": 0, "max_value": 255, "bit_len": 8},
      "twos_complement": false, "alignment": "NATURAL",
      "bit_shift": {"shift_dir": "NONE", "shift_amount": 0}
    }
  ]
}
```

The JSON object for the SWI opcode group is very simple and has very little in the way of new material. I'll let you look over it on your own. Be sure to review it and understand it before you move on to the next section.

## Format-18: Unconditional Branch - B

### Bit Breakdown: Software Interrupt

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
-----															
	1		1		1		0		0		Offset11				

### Details:

Offset11 = 11-bit Two's complement address  
Cond. Codes = Condition codes are not set by this opcode group.  
Notes = This instruction performs a PC-relative Branch. The branch offset must take into account the prefetch operation, which causes the PC to be 1 word (4 bytes) ahead of the current instruction. The address specified by label is a full 12-bit two's complement encoded address, but must always be halfword aligned (i.e. bit 0 set to 0), since the assembler places label >> 1 in the Offset11 field.

### Description:

#### Thumb-1 Code      Action

B label      Branch PC relative +/- Offset11 << 1, where label is PC +/- 2048 bytes.

### Examples:

Endian	Code	Binary Rep	Hex Rep
Little	B `label_	10011010 11100111	0x9A E7
Big	B `label_	11100111 10011010	0XE7 9A
Little	B `over	00101011 11100000	0x2B E0

Big	B `over	11100000 00101011	0xE0 2B
-----	---------	----------------------	---------

#### JSON Representation:

```
{
  "obj_name": "is_op_code",
  "op_code_name": "B",
  "index": 80,
  "arg_len": 1,
  "arg_separator": ",",
  "bit_rep": {"bit_string": "11100", "bit_int": 28, "bit_len": 5},
  "bit_series": {"bit_start": 11, "bit_stop": 15, "bit_len": 5},
  "args": [
    {
      "obj_name": "is_op_code_arg",
      "arg_index": 0,
      "bit_index": 0,
      "is_entry_types": ["Number", "LabelRef"],
      "is_arg_type": "Offset11",
      "bit_series": {"bit_start": 0, "bit_stop": 10, "bit_len": 11},
      "num_range": {"min_value": -2048, "max_value": 2048, "bit_len": 12, "twos_complement": true, "handle_prefetch": true,
      "use_halfword_offset": true, "alignment": "HALFWORD"},
      "bit_shift": {"shift_dir": "RIGHT", "shift_amount": 1}
    }
  ]
}
```

The JSON data entry for the B instruction is listed previously. There's nothing new really to review with regard to the entry. Pay special attention to the alignment, the shift amount, and the two's complement flag. Have you noticed a pattern with regard to the alignment and the shift amount? See if you can find one. In the next, and last, section we'll review the BL opcode. Let's take a look!

## Format-19: Long Branch with Link - BL

Bit Breakdown: Long Branch with Link

15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

| 1 | 1 | 1 | 1 | H | Offset |

**Details:**

H = Low/high offset bit

H = 0 -> Offset high

H = 1 -> Offset low

Offset = Long branch and link offset high/low

Cond. Codes = Condition codes are not set by this opcode group.

Notes = This format specifies a long branch with link. The assembler splits the 23-bit two's complement half word offset specified by the label into two 11-bit halves, ignoring bit 0, which must be zero, and creates two Thumb-1 instructions.

= Instruction 1 (H = 0): In the first instruction of the Offset field contains the upper 11 bits of the target address. This is shifted left by 12 bits and added to the current PC address. The resulting address is placed in LR.

= Instruction 2 (H = 1): In the second instruction the Offset field contains an 11-bit representation, lower half of the target address. This is shifted left by 1 bit and added to LR. The LR register, which now contains the full 23-bit address, is places in PC, the address of the instruction following the BL is places in LR and bit 0 of LR is set.

= The branch offset must take account of the prefetch operation, which causes the PC to be 1 word (4 bytes) ahead of the current instruction.

**Description:**

H	Thumb-1 Code	Action
0	BL label	$LR = PC + OffsetHigh \ll 12$
1	BL label	$temp = \text{next instruction address including pre-fetch}$ $PC = LR + OffsetLow \ll 1$ $LR = temp   1$

### Examples:

Endian	Code	Binary Rep	Hex Rep
Little	BL `label_	11111111 11110111	0xFF F7
	;LINE 2	10011000 11111111	0x98 FF
Big	BL `label_	11110111 11111111	0xF7 FF
	;LINE 2	11111111 10011000	0xFF 98
Little	BL `over	00000000 11110000	0x00 F0
	;LINE 2	00101000 11110000	0x28 F8
Big	BL `over	11110000 00000000	0xF0 00
	;LINE 2	11111000 00101000	0xF8 28

### JSON Representation:

```
{  
    "obj_name": "is_op_code",  
    "op_code_name": "BL",  
    "index": 81,  
    "arg_len": 1,  
    "arg_separator": ",",  
    "bit_rep": {"bit_string": "11110", "bit_int": 30, "bit_len": 5},  
    "bit_series": {"bit_start": 11, "bit_stop": 15, "bit_len": 5},  
    "args": [  
        {  
            "obj_name": "is_op_code_arg",  
            "arg_index": 0,  
            "bit_index": 0,  
            "bit_len": 1,  
            "bit_start": 0, "bit_stop": 1  
        }  
    ]  
}
```

```

    "is_entry_types": ["Number", "LabelRef"],
    "is_arg_type": "Offset11",
    "bit_series": {"bit_start": 0, "bit_stop": 10, "bit_len": 11},
    "num_range": {"min_value": 0, "max_value": 8388606, "bit_len": 23,
      "twos_complement": true, "handle_prefetch": true,
      "use_halfword_offset": true, "alignment": "HALFWORD"},
      "bit_shift": {"shift_dir": "NONE", "shift_amount": -1}
    }
  ]
}

```

The `BL` instruction is different from ALL of the other instructions in the Thumb-1 instruction set. Did you notice why that is? Well, if you thought about the fact that its binary representation takes not one but two lines of binary code. This is the one and only instruction in the Thumb-1 instruction set that does this and thank goodness too because it's a bit awkward to expand the binary representation by one line for each `BL` instruction.

Take a moment to let all the instruction code knowledge you just reviewed to sink in. Don't worry if it doesn't resonate with you right away. After all, it takes some experience with assembly code to be able to see the forest for the trees. No worries, we'll get there soon enough.

## Chapter Conclusion

In this chapter we covered a lot of material regarding the ARM Thumb-1 instruction set wrapping up the review that began in the previous chapter. There were a lot of interesting opcodes, some we've seen before in one form or another. And we even encountered a few of the Thumb-1 instructions set's caveats.

**Format-10:** Load/Store Halfword – `STRH`, `LDRH`: Here we reviewed load and store opcodes for half-word data.

**Format-11:** SP-Relative Load/Store – `STR`, `LDR`: In this section we reviewed load and store opcodes that are addresses relative to the SP register's value.

**Format-12:** Load Address – ADD: In this section we took a look at using the ADD instruction to calculate and load an address value into a register.

**Format-13:** Add Offset to Stack Pointer – ADD: Another ADD opcode permutation that uses an offset to the SP register's value.

**Format-14:** Push/Pop Registers – PUSH, POP: A quick review of the instructions used to push and pop values off of the stack.

**Format-15:** Multiple Load/Store – STMIA, LDMIA: In this section, we took a look at the instructions used to store and load multiple values.

**Format-16:** Conditional Branch – BEQ, BNE, BCS, BCC, BMI, BPL, BVS, BVC, BHI, BLS, BGE, BLT, BGT, BLE: Branching is an important aspect of assembly programming and as such there are a lot of different ways to branch based on condition flags.

**Format-17:** Software Interrupt – SWI: A quick review of the software interrupt opcode.

**Format-18:** Unconditional Branch – B: An important, and often used, branching instruction that is not based on offsets.

**Format-19:** Long Branch with Link – BL: The last opcode in the ARM Thumb-1 instruction set that supports long branching by using an 11-bit value that can be used as the upper or lower bits used in calculating the branch destination address.

Congrats! You made it through a fairly intense review of the Thumb-1 instruction set. We won't be writing assembly programs just yet, first we have to build our assembler! Before we get into the details of how the GenAsm assembler works I want to review some of the desired functionality of the assembler. As such we'll begin reviewing the preprocessor directives, and assembly directives that are supported by our project. Preprocessor directives and assembly directives are used along-side normal instruction set opcodes to create a functioning program.

# Chapter 5: Preprocessor Directives

In this chapter we'll review the GenAsm assembler's preprocessor directives. A preprocessor directive, as you might have imagined, is an assembly directive that runs before any real file processing takes place. The benefit of this type of directive is that we can alter the contents of the assembly source code file before it's processed by the lexer.

We'll cover the steps to process and assembly source code file in detail in just a bit but, just to be clear, the lexer is the first step in processing a text-based assembly source code file. The lexer breaks the lines of text into lines of "artifacts" or pieces of text. The lexer determines how to break up the text based on a set of syntax rules. I digress, we'll get to that topic in more detail soon. The preprocessor directives that are supported by the GenAsm assembler are as follows:

- Include Binary File: \$INCBIN, \$FLPINCBIN
- Include Assembly File: \$INCASM
- Non-Operation: \$NOP
- Include String Data: \$STRING, \$FLIPSTRING

We'll start things off with a review of the supported preprocessor directives for each type in the previous list. We'll take a look at an example of each directive in action, and we'll also take a look at the results of the preprocessor directives once it's been handled by the assembler. I should take a moment to mention that the relative file paths used in the include preprocessor directives are designed to be relative to the GenAsm project's working directory setting. Let's have a look!

## Include Binary File Preprocessor Directives: \$INCBIN, \$FLPINCBIN

The first group of preprocessor directives we'll look at are the include binary file directives. There are two entries in this group, \$INCBIN and \$FLPINCBIN. Notice that preprocessor directives start with a '\$' and are distinguishable from other opcodes and directives in this way. Recall that I mentioned earlier the assembler was forward looking and that each artifact type, preprocessor directive, regular directive, opcode, is known from the first character in the artifact.

*Listing 5-1: Assembly Preprocessor Directive Source Example - \$INCBIN*

```
1 program_start
2 ;no need to flip
3 ;+++++
4 ;$INCBIN |.\cfg\THUMB\INCBIN\gba_arm2thumb_jump_addr_00F4.bin|
5 ;+++++
```

An assembly source code snippet showing the use of the \$INCBIN preprocessor directive pointing to a binary file.

In the next listing we'll see the similar use case of the flipped include binary preprocessor directive.

*Listing 5-2: Assembly Preprocessor Directive Source Example - \$FLPINCBIN*

```
1 program_start
2 ;already little endian, need to flip
3 ;+++++
```

```
4 ;$FLPINCBIN |.\cfg\THUMB\INCBIN\gba_arm2thumb_jump_addr_00F4.bin|
5 ;+++++
```

An assembly source code snippet showing the use of the \$FLPINCBIN preprocessor directive pointing to a binary file which will apply an endian flip on the included binary data.

You should be asking yourself, “Why are there two versions of the include binary preprocessor directive?” There is a hint in the comments of the previous listing. Can you see it? The note in the comments says, “already little endian, need to flip.” This is because the assembler expects binary data, numbers, etc., to be in big-endian format by default and by performing an endian flip achieve little-endian format. It just so happens that the endian flip process is cyclical.

Thus, if you started with little-endian binary data and you entered it into your assembly source code file it would have a little-endian encoding when a big-endian encoding is expected. Subsequently, when the binary data gets converted for the little-endian binary output, this data would be converted to big-endian format. Not little-endian format as expected. In short converting binary data endianness is cyclical so you must make sure you have your data in the correct format before applying the conversion. Let’s take a look at the listing output generated by the preprocessor directive, \$FLPINCBIN, listed previously.

*Listing 5-3: Assembly Preprocessor Directive Listing Example - \$FLPINCBIN*

```
//Header
1 ;+++++
2 ;$FLPINCBIN |.\cfg\THUMB\INCBIN\gba_header.bin|
3 ;Found file with byte count: 232, word count: 116, and padding on: false

//BIG-ENDIAN
1 0x0038 @DCHW #0x0038 ;index: 0000 address: 0x0000
2 0xEA00 @DCHW #0xEA00 ;index: 0001 address: 0x0002
3 0x0000 @DCHW #0x0000 ;index: 0002 address: 0x0004
4 0x0000 @DCHW #0x0000 ;index: 0003 address: 0x0006

//LITTLE-ENDIAN
```

```
1 0x3800 @DCHW #0x0038 ;index: 0000 address: 0x0000
2 0x00EA @DCHW #0xEA00 ;index: 0001 address: 0x0002
3 0x0000 @DCHW #0x0000 ;index: 0002 address: 0x0004
4 0x0000 @DCHW #0x0000 ;index: 0003 address: 0x0006
```

Assembly listing snippets showing the results of the preprocessor directive, `$FLPINCBIN`, for both the big- and little-endian listing output. Notice that the little-endian binary output is flipped when compared to the big-endian binary output.

The first thing I want you to look at is the header entry in the previous listing. As you can see the target file is scanned to determine how many 2-byte entries are needed to list the file and the information is placed in-line, ahead, of the file contents. Take a moment to notice that the preprocessor directive actually gets converted into a data directive, `@DCHW`, which gets processed later on as part of the normal assembly source code processing procedure.

To help keep track of the generated binary source code lines the data directive is commented with an index, half-word offset, and a memory address. This will help you track down any issues between the included binary file and the generated data directives. Up next, we'll take a look at a preprocessor directive that is used to include assembly source code in much the same way we handled including binary data.

## **Include Assembly File Preprocessor Directive: \$INCASM**

The next preprocessor directive we'll look at is used to include assembly source code from another file, in-line, in your current assembly source code file. This functions in much the same way as the `$FLINCBIN` and `$INCBIN` preprocessor directives except that instead of converting binary data to `@DCHW` data directives we simply insert the additional assembly source code at the location of the preprocessor directive. Let's take a look at the assembly source code syntax for this preprocessor directive.

*Listing 5-4: Assembly Preprocessor Directive Source Example - \$INCASM*

```
1 ;+++++  
2 ;$INCASM |.\cfg\THUMB\TESTS\test_asm_incasm.txt|  
3 ;+++++
```

An assembly source code example of the \$INCASM preprocessor directive.

If you're wondering why all the preprocessor directives are on commented lines you've been paying attention. The reason for this is because we don't want the directive to be handled by anything other than the preprocessor. By commenting out the line the assembler will ignore it. It is also a good time to note that the data directives generated by these preprocessor directives are placed after the original commented line. This helps with readability and understanding of the assembly code after it leaves the preprocessor. In the next section we'll take a look at the included assembly source code after it's handled by the preprocessor.

*Listing 5-5: Assembly Preprocessor Directive Listing Example - \$INCASM*

```
//HEADER  
1 ;+++++  
2 ;$INCASM |.\cfg\THUMB\INCBIN\test_asm_incasm.txt|  
3 ;Found file with line count: 4, byte count: 86  
  
//BIG-ENDIAN  
1 0x4BD3 LDR R3, [PC, #844]  
2 0x5193 STR R3, [R2, R6]  
3 0x5DC2 LDRB R2, [R0, R7]  
4 0x521C STRH R4, [R3, R0]  
  
//LITTLE-ENDIAN  
1 0xD34B LDR R3, [PC, #844]  
2 0x9351 STR R3, [R2, R6]  
3 0xC25D LDRB R2, [R0, R7]  
4 0x1C52 STRH R4, [R3, R0]
```

Assembly listing snippets showing the results of the preprocessor directive, \$INCASM, for both the big- and little-endian listing output. Notice that the little-endian binary output is flipped when compared to the big-endian binary output.

In the previous listing, under the “header” section you can see how the Genasm processes the command. A summary of the line count and byte count found in the included assembly file are added to the original source code. In the next two sections there are examples of the added assembly source code for both big- and little-endian binary encodings. You can use this preprocessor directive to add assembly source code from multiple files to your main project file. This can help you manage your code a little better if you find your project getting a little too large.

## No-Operation Preprocessor Directive: \$NOP

In this section we'll take a look at the \$NOP preprocessor directive. This directive is used to inject an innocuous instruction at the location of the directive. Let's take a look at this preprocessor directive in action.

*Listing 5-6: Assembly Preprocessor Directive Source Example - \$NOP*

```
1 LDR R3, [PC, #844]  
2 STR R3, [R2, R6]  
3 ;$NOP  
4 LDRB R2, [R0, R7]  
5 STRH R4, [R3, R0]
```

An assembly source code example of the \$NOP preprocessor directive.

As you can see from the example you use this directive as you would any other opcode. I tend to comment out the preprocessor directive, but this isn't strictly required.

*Listing 5-7: Assembly Preprocessor Directive Listing Example - \$NOP*

```
//BIG-ENDIAN  
1 0x46C0 MOV R8, R8      ;NOP preprocessor directive  
  
//LITTLE-ENDIAN  
1 0xC046 MOV R8, R8      ;NOP preprocessor directive
```

An assembly listing example showing the results of the \$NOP preprocessor directive.

Let's look at how the preprocessor directive is handled by the assembler. As shown in the previous listing, for both endian encodings, the \$NOP preprocessor directive is converted to an innocuous instruction, copying a register's value onto itself. Can you think of a need for this particular directive? If you thought about spacing out your assembly source code, you're on the right track. How do you "move" an instruction up in memory so that it is word aligned.

Well, the \$NOP preprocessor directive can help with this a little bit as it adds an innocuous instruction into the source code effectively shifting all subsequent instructions by two bytes. This can really come in handy when you want to control the memory location of certain instructions and/or values. In the next section we'll wrap up the preprocessor directives by reviewing a pair of directives that convert a string into a series of assembler data directives.

## Include String Data Preprocessor Directive: \$FLIPSTRING, \$STRING

The \$FLIPSTRING and \$STRING preprocessor directives are used to convert a string to a series of data directives that express the string as a series of shorts.

*Listing 5-8: Assembly Preprocessor Directive Source Example - \$FLIPSTRING, \$STRING*

```
1 ;+++++  
2 ;$STRING |Hello World!|  
3 ;+++++  
4  
5 ;+++++  
6 ;$FLIPSTRING |Hello World!|  
7 ;+++++
```

An assembly source code example demonstrating the use of the \$FLIPSTRING and \$STRING preprocessor directives. Take a moment to notice that we're encoding both

big-endian and little-endian because the `$FLIPSTRING` directive will flip the binary encoding's endianness.

In the previous listing we see both directives in use in one snippet. Note that this preprocessor directive section contains a good example of the cyclical nature of flipping the endian encoding. Keep an eye out for it.

*Listing 5-9: Assembly Preprocessor Directive Listing Example - `$FLIPSTRING`, `$STRING`*

```
//STRING DIRECTIVE
01 ;+++++
02 ;$STRING |Hello World!
03 ;Found string byte count: 14, word count: 7, and padding on: true
04
05 @DCHW #0x4865      ;index: 0000      H e
06 @DCHW #0x6C6C      ;index: 0002      l l
07 @DCHW #0x6F20      ;index: 0004      o
08 @DCHW #0x576F      ;index: 0006      W o
09 @DCHW #0x726C      ;index: 0008      r l
10 @DCHW #0x6421      ;index: 0010      d !
11 @DCHW #0xFFFF      ;index: 0012
12
13 ;+++++
14

//FLIP-STRING DIRECTIVE
15 ;+++++
16 ;$FLIPSTRING |Hello World!
17 ;Found string byte count: 14, word count: 7, and padding on: true
18
19 @DCHW #0x6548      ;index: 0000      H e
20 @DCHW #0x6C6C      ;index: 0002      l l
21 @DCHW #0x206F      ;index: 0004      o
22 @DCHW #0x6F57      ;index: 0006      W o
23 @DCHW #0x6C72      ;index: 0008      r l
24 @DCHW #0x2164      ;index: 0010      d !
25 @DCHW #0xFFFF      ;index: 0012
26
27 ;+++++
```

An assembly listing example for the \$FLIPSTRING and \$STRING preprocessor directive showing the header and both directive's output.

An example of the processed directives is shown in the previous listing.

Note that each preprocessor directive call has its own header that indicates the calculated byte count, the word count, and if padding is on. The padding option is on by default. This will add blank values to the end of the provided string argument to ensure that the binary encoding is half-word aligned, aligned to a 2-bytes boundary. Also take a moment to note that the string data is suffixed with a null value, /0, as you might see in a C program. This technique is often referred to as a C-style string.

I should take a moment to mention that the reason why we're not looking at any JSON objects for the supported preprocessor directives is that they are applied to the assembly source code file before any instruction set data. This means that these directives are expressed entirely in code because they have to run before the JSON data files are loaded.

## Chapter Conclusion

In this chapter we covered all the supported preprocessor directives. There are some blaring omissions to the preprocessor's functionality supported by the GenAsm assembler namely if-else logic. That's fine, we have quite a bit of functionality and as you've seen we can leverage it to create a program that runs on a GameBoy Advance emulator. Not too shabby. Let's review the preprocessor directives we've reviewed in this chapter.

- 1. Include Binary File Preprocessor Directives:** \$INCBIN, \$FLPINCBIN: A set of preprocessor directives that are used to add existing binary instructions to assembly source code in the form of data directives.
- 2. Include Assembly File Preprocessor Directive:** \$INCASM: A second method for adding instructions to an assembly program. This version however, adds assembly code not binary code as with the \$INCBIN,

`$FLPINCBIN` directives.

3. **No-Operation Preprocessor Directive:** `$NOP`: A preprocessor directive that can be used to insert an innocuous instruction into an assembly program.
4. **Include String Data Preprocessor Directive:** `$FLIPSTRING`, `$STRING`: A set of preprocessor directives that are used to convert a string to a series of data directives, inserted into the assembly source code.

That brings us to the end of the preprocessor directive review. In the next chapter we'll take a look at the set of directives supported by the GenAsm assembler. From there we'll move on to review some code and see how the assembler actually works. It's going to be a challenging and interesting journey, let's get started!

# Chapter 6: Standard Directives

In Chapter 6 we'll get one step closer to working on the code that powers the GenAsm assembler by taking a look at the assembler's supported directives. In general, directives are instructions used by the assembler to help the assembly process by providing ways to set the data at specific memory locations and to improve program readability. The definition holds true for our assembler as well. I'll quickly list the assembler's supported directives here and we'll review each one in detail throughout the chapter.

- **@DCHW:** A data directive used to write two bytes of data at the current memory location.
- **@FLPDCHW:** A data directive used to write two bytes of data, with flipped endianness, at the current memory location.
- **@DCWBF:** A data directive used to write the first byte of a two-byte value as a half-word, at the current memory location.
- **@DCWBS:** A data directive used to write the second byte of a two-byte value as a half-word, at the current memory location.

- **@DCB:** A data directive used to write one byte of data as a half-word at the current memory location.
- **@EQU:** A label directive used to set a label to a specific value, creating a label value entry that can be referenced by subsequent assembly instructions.
- **@ORG:** A memory directive used to set the starting memory address to be used in all listing and memory address calculations.
- **@TTL:** An area definition directive used to define a program's title.
- **@SUBT:** A program definition directive used to define a program's subtitle.
- **@ENTRY:** An area definition directive used to define the start of an area.
- **@END:** An area definition directive used to define the end of an area.
- **@AREA:** An area definition directive used to indicate the start of an area definition.
- **@READONLY:** An area definition directive used to indicate if the area is read-only. Not currently in use.
- **@READWRITE:** An area definition directive used to indicate if the area is read-write. Not currently in use.
- **@CODE:** An area definition directive used to indicate if the area is a code area and not a data area.
- **@DATA:** An area definition directive used to indicate if the area is a data area and not a code area.

Take a look at each directive in the list and try to imagine it in use. When writing assembly programs, you are going to be concerned with memory locations and the value stored in a given memory location in ways

that you wouldn't when working with higher level languages. Note that each directive starts with the '@' character. As we mentioned earlier the assembler is forward looking meaning that each type of token can be determined from the first character of the token.

If you recall the assembler's preprocessor directives all started with a '\$' character. Keep this notion in mind as you learn about the different types of tokens that can be used in an assembly source code file. We'll cover it in more detail soon, but for our purposes here a token is a recognized string object that has a certain type along with other properties.

## Data Directives 2-Byte Values: @DCHW, @FLPDCHW

The first group of directives we'll review are the data directives that are used to write a half-word value to the current memory address. I should make the distinction that all data directives write 2-byte values because the ARM Thumb-1 instruction set has a 2-byte bit width for each instruction, but this doesn't mean that a 2-byte value is actually being written. There are other data directives that are designed to write a 1-byte value that is padded with 0's to take up the requisite 16-bits per line.

*Listing 6-1: Assembly Directive Source Example - @DCHW, @FLPDCHW*

```
01 ;;;;;;;;;;;;;
02
03         @DCHW    0x0000
04         @DCHW    0x0000
05         @DCHW    0x0000
06         @DCHW    0x0000
07 bitmap_font      ;1 Bit per pixel font
08         @FLPDCHW 0x183C
09         @FLPDCHW 0x6666
10         @FLPDCHW 0x7E66
11         @FLPDCHW 0x2400 ;A
```

An assembly source code example from the TEXT\_B\_16BitXfer program demonstrating use of the @DCHW and @FLPDCHW directives.

In the previous listing we can see the use of both 2-byte data directives in action. The relative path to the assembly source code file that the snippet was pulled from are as follows.

```
//SOURCE CODE FILE  
.cfg\THUMB\TESTS\TEST_B_16BitXfer\genasm_source.txt  
  
//LISTING OUTPUT FILES  
.cfg\THUMB\OUTPUT\TEST_B_16BitXfer\  
output_assembly_listing_endian_big.list  
  
.cfg\THUMB\OUTPUT\TEST_B_16BitXfer\  
output_assembly_listing_endian_lil.list
```

Note that the relative path is based on the project's working directory setting. We have the directory structure setup such that the program source is separate from the assembler's output. Take a moment to look around at the different test programs. We'll be working with each one in detail after we've completed the assembler's source code review. Notice that, in the previously listed example, we are writing both endian flipped and direct values. Can you think of a reason why we would need to do this? Hint, think back to our preprocessor directive review in Chapter 5, specifically the data inclusion directives.

Have you figured it out? Similar to the \$INCBIN preprocessor directive the @DCHW directive has an endian flipped version, @FLPDCHW. This allows us to use data that is in little-endian format, flipping its endianness and writing the correct 2-byte value at the current memory location. When talking about memory locations you have to realize that each line in an assembly program can be referenced at a specific memory location. Think about the program residing in ram at a specific memory location then think about how each subsequent instruction in the program is 2-bytes farther away from the program's starting memory location.

*Listing 6-2: Assembly Directive Listing Example - @DCHW, @FLPDCHW*

```
01 ;;;;;;;;;;  
02  
03 ... 0x080001E8 ... 0x0000 @DCHW 0x0000
```

```

04 ... 0x080001EA ... 0x0000 @DCHW 0x0000
05 ... 0x080001EC ... 0x0000 @DCHW 0x0000
06 ... 0x080001EE ... 0x0000 @DCHW 0x0000
07           bitmap_font ;1 Bit per ...
08 ... 0x080001F0 ... 0x3C18 @FLPDCHW 0x183C
09 ... 0x080001F2 ... 0x6666 @FLPDCHW 0x6666
10 ... 0x080001F4 ... 0x667E @FLPDCHW 0x7E66
11 ... 0x080001F6 ... 0x0024 @FLPDCHW 0x2400 ;A

```

An assembly listing example from the TEXT\_B\_16BitXfer program demonstrating the results of using the @DCHW and @FLPDCHW directives.

The first thing I want you to notice in the previously shown listing example is that only the lines of assembly source code that end up in the final binary output file are given a memory address and value. We've shortened the standard listing line a little bit and excluded the binary value representations for brevity. When we mention that a certain memory address has a specific value, we mean the binary value written at that address, location, in the program. In the next listing we'll take a look at the JSON object for this group of directives.

*Listing 6-3: Assembly Directive JSON Data Entry Example - @DCHW*

```

01 {
02   "obj_name": "is_directive",
03   "directive_name": "@DCHW",
04   "arg_len": 1,
05   "args": [
06     {
07       "obj_name": "is_directive_arg",
08       "arg_index": 0,
09       "bit_index": 0,
10       "is_entry_types": ["Number", "LabelRef"],
11       "is_arg_type": "Number",
12       "bit_series": {"bit_start": 0, "bit_stop": 15, "bit_len": 16},
13       "num_range": {"min_value": 0, "max_value": 65536, "bit_len": 16},
14       "twos_complement": false, "alignment": "HALFWORD"},
15       "bit_shift": {"shift_dir": "NONE", "shift_amount": -1}
16     }
16 ],

```

```
17    "description": "The DCHW directive is used to write data, one  
halfword, directly into the data AREA of an assembly file."  
18 }
```

The JSON data entry for the @DCHW directive.

The JSON object for the @DCHW directive is shown in the previous listing. I've excluded the @FLPDCHW JSON object because it's almost identical and would be considered redundant. Take a moment to look over the entry. Does anything stand out? If you thought that the data entry looked a lot like the entries we've seen during the opcode review chapters you're on the right track. Directives take arguments in the same way that opcode's do. As such their JSON objects look similar because they share a lot of the same attributes. Note that the JSON objects for the assembler's directives are found in the following data file:

```
.cfg\THUMB\is_directives.json
```

Take a moment to look over the entry and make sure you understand it before moving on. There aren't any new attributes or sub-objects with regard to the opcode JSON data entries we've already seen so the structure of the object should be very familiar. In the next section we'll take a look at 1-byte data directives.

## **Data Directives 1-Byte Values: @DCB, @DCWBF, @DCWBS**

In this section we'll review the data directives that are used to write 1-byte values as half-words at the current memory location. These directives are similar to the 2-byte data directives we've just reviewed. The main difference is that this group of directives writes a 1-byte value with a 1-byte, zeroed, padding value. In this way they are considered 1-byte data directives because only one of the 2-bytes used for an assembly instruction contains a real value. Let's take a look at some assembly code!

*Listing 6-4: Assembly Directive Source Example - @DCB, @DCWBF, @DCWBS*

```
//Assembly Source
```

```
1 @DCB      5
2 @DCWBS   0x01AC
3 @DCWBF   0x01AC
```

An example assembly source code demonstrating 1-byte data directives.

In the previous listing we've demonstrated the source code for the 1-byte directives in this group. Take a look at the entries listed. Short and sweet. Now let's take a look at how the assembler processes these directives.

*Listing 6-5: Assembly Directive Listing Example - @DCB, @DCWBF, @DCWBS*

```
//BIG-ENDIAN
1 ... 0x0000013C ... 0x0005 @DCB      5
2 ... 0x0000013E ... 0x0001 @DCWBS   0x01AC
3 ... 0x00000140 ... 0x00AC @DCWBF   0x01AC

//LITTLE-ENDIAN
4 ... 0x0000013C ... 0x0500 @DCB      5
5 ... 0x0000013E ... 0x0100 @DCWBS   0x01AC
6 ... 0x00000140 ... 0x0000 @DCWBF   0x01AC
```

An example assembly listing snippet showing the big and little-endian output generated by the assembler for the given 1-byte directive.

The resulting assembler output for these directives is shown previously. Notice that for each directive listed a 1-byte value is written along with a zero padded byte. Also take a moment to notice that the @DCWBF and @DCWBS data directives write the first or second byte, respectively, in the big-endian encoding of the value 0x01AC. In either case the resulting binary output has the proper endian encoding showing the first or second byte in the proper position of the hexadecimal listing output, lines 2, 3, and 5, 6. The previously listed examples are from the following test program and associated output files listed here.

```
//SOURCE CODE FILE
.\cfg\THUMB\TESTS\TEST_N_AsmChecks\genasm_source.txt
```

```
//LISTING OUTPUT FILES
.\cfg\THUMB\OUTPUT\TEST_N_AsmChecks\
output_assembly_listing_endian_big.list

.\cfg\THUMB\OUTPUT\TEST_N_AsmChecks\
output_assembly_listing_endian_lil.list
```

Take a look at the assembly source code file and associated output files before moving on to the next section. Let's quickly look at the JSON object for this group of directives.

*Listing 6-6: Assembly Directive JSON Data Entry Example - @DCB*

```
01 {
02     "obj_name": "is_directive",
03     "directive_name": "@DCB",
04     "arg_len": 1,
05     "args": [
06         {
07             "obj_name": "is_directive_arg",
08             "arg_index": 0,
09             "bit_index": 0,
10             "is_entry_types": ["Number", "LabelRef"],
11             "is_arg_type": "Number",
12             "bit_series": {"bit_start": 0, "bit_stop": 15, "bit_len": 8},
13             "num_range": {"min_value": 0, "max_value": 255, "bit_len": 8, "twos_complement": false, "alignment": "HALFWORD"},
14             "bit_shift": {"shift_dir": "NONE", "shift_amount": -1}
15         }
16     ],
17     "description": "The DCB directive is used to write data, one byte written as a halfword, directly into the data AREA of an assembly file."
18 }
```

The JSON data entry for the @DCB directive.

The JSON object for the single byte group of directives is listed previously. There's nothing special about this entry. It should look familiar to some of the opcode and directive entries you've already seen. The JSON entries for data directives @DCWBF and @DCWBS are very similar to the previous listing so we won't include them here due to redundancy. Take a moment to look at the JSON entry it and understand it before you move on to the next section.

## Label and Memory Directives: @EQU, @ORG

In this section of directives for us to review we'll take a look at the GenAsm assembler's label and memory directives. In this case there are two more directives for use to take a look at. We'll look at the @EQU directive first followed by the @ORG directive.

*Listing 6-7: Assembly Directive Source Example - @EQU*

```
1 test      @EQU 2
```

The assembly source code example is very simple and shows the @EQU directive in use.

The simplicity of the @EQU directive can be misleading it's a very useful and powerful directive. In the previously listed example, we're creating a new value label, test, and assigning it a value of 2. Now we can refer to the value by its label and use it almost like we would a simple variable in a higher-level programming language.

*Listing 6-8: Assembly Directive Source Example - @EQU*

```
1 ;===== SYMBOL TABLE =====
2 ...
3 ...
4 ...
5 ;Name: test LineNumAbs: 000000004 LineNumActive: 000000000 AddressHex:
Value: 2 EmptyLineLabel: false IsLabel: false IsStaticValue: true
```

The assembly listing source code example shows the symbol table entry for the @EQU directive.

In the previous listing we're taking a look at a line from the symbol table that's printed at the end of the assembler's listing output files. Notice that the entry has the IsLabel attribute set to false and the IsStaticValue attribute set to true. Can you see why the AddressHex attribute is left blank for this symbol table entry? If you thought that it was left blank because the "test" symbol is not actually part of the final binary program you thought right.

This is the signature of a value label created using the @EQU directive. Think about how useful it is to create labels with associated numeric values to use throughout the assembly source code. The previous examples are from the following test program and associated output files listed here.

```
//SOURCE CODE FILE
.\cfg\THUMB\TESTS\TEST_N_AsmChecks\genasm_source.txt

//LISTING OUTPUT FILES
.\cfg\THUMB\OUTPUT\TEST_N_AsmChecks\
output_assembly_listing_endian_big.list

.\cfg\THUMB\OUTPUT\TEST_N_AsmChecks\
output_assembly_listing_endian_lil.list
```

Next, let's look at this directive's JSON object. There's not much in the way of new material here but, I'll list it anyhow to be thorough.

*Listing 6-9: Assembly Directive JSON Data Entry Example - @EQU*

```
01 {
02   "obj_name": "is_directive",
03   "directive_name": "@EQU",
04   "arg_len": 1,
05   "args": [
06     {
```

```

07         "obj_name": "is_directive_arg",
08         "arg_index": 0,
09         "bit_index": 0,
10         "is_entry_types": ["Number"],
11         "is_arg_type": "Number",
12         "bit_series": {"bit_start": 0, "bit_stop": 15, "bit_len": 16},
13             "num_range": {"min_value": 0, "max_value": 65536, "bit_len": 16, "twos_complement": false, "alignment": "WORD"},
14             "bit_shift": {"shift_dir": "NONE", "shift_amount": -1}
15         }
16     ],
17     "description": "The EQU directive is used to store a 16-bit value in the symbol table for value lookup."
18 }
```

The JSON data entry for the @EQU directive. Note that the directive is used in conjunction with a label, but the label is not part of the directive argument list.

Next, we'll take a look at the @ORG directive. The @ORG directive is used to set the starting memory address for the given assembly source code file. The reason why this is useful, very useful in fact, has more to do with the different hardware executing your program. For instance, the GameBoy Advance emulator, and hardware, will execute your program from a specific memory location where the current program is copied for execution. In order to make our memory addresses and associated values match up with the hardware we need to tell the assembler what the starting memory address is. This address is then used in calculations regarding memory locations. Let's take a look at the directive in action.

#### *[Listing 6-10: Assembly Directive Source Example - @ORG](#)*

```
1 @ORG 0x08000000
```

The assembly source code example is very simple and shows the @ORG directive in use.

The @ORG directive is very simple to use, all you need to do is provide a numeric argument that is the starting memory address used by your target hardware. In this case the GameBoy Advance emulator uses memory location, 0x08000000, as the starting point for executing the program provided. This means that the binary representation of the program gets copied to this location before execution begins. Let's take a look at how this directive affects the other lines in the assembly source code file.

*Listing 6-11: Assembly Directive Listing Example - @ORG*

```
1 @ORG 0x08000000
2
3 ;need 122 16bit entries to match GBA header
4 ;already little endian, need to flip
5 ;+++++;+++++;+++++;+++++;+++++
6 ;$FLPINCBIN |.\cfg\THUMB\INCBIN\gba_header.bin|
7 ;Found file with byte count: 232, word count: 116, and padding on: false
8
9 ... 0x08000000 ... 0x3800 @DCHW #0x0038 ;index: 0000 address: 0x0000
```

An assembly listing example showing the use of the @ORG directive and the effect on subsequent memory addresses.

Notice that the first line has the @ORG directive which sets the starting memory address to 0x08000000. Now, if you direct your attention to line 9, the hexadecimal representation of the memory address has the same value as that set by the @ORG directive. Also note that this is the first line that has a memory address as all previous lines in this listing do not end up in the binary representation of the assembly source code. The source code and output files used in this directive review are as follows.

```
//SOURCE CODE FILE
.\cfg\THUMB\TESTS\TEST_B_16BitXfer\genasm_source.txt

//LISTING OUTPUT FILES
.\cfg\THUMB\OUTPUT\TEST_B_16BitXfer\
output_assembly_listing_endian_big.list
```

```
.\\cfg\\THUMB\\OUTPUT\\TEST_B_16BitXfer\\  
output_assembly_listing_endian_lil.list
```

To wrap up our review of the @ORG directive we'll quickly look at the directive's JSON object. This directive takes a number as an argument, much like previous directives we've reviewed. The directive's JSON object will look similar to previous entries we've reviewed in this chapter.

*Listing 6-12: Assembly Directive JSON Data Entry Example - @ORG*

```
01 {  
02     "obj_name": "is_directive",  
03     "directive_name": "@ORG",  
04     "arg_len": 1,  
05     "args": [  
06         {  
07             "obj_name": "is_directive_arg",  
08             "arg_index": 0,  
09             "bit_index": 0,  
10             "is_entry_types": ["Number"],  
11             "is_arg_type": "Number",  
12             "bit_series": {"bit_start": 0, "bit_stop": 31, "bit_len":  
32},  
13                 "num_range": {"min_value": 0, "max_value": 2147483647,  
"bit_len": 32, "twos_complement": false, "alignment": "WORD"},  
14                 "bit_shift": {"shift_dir": "NONE", "shift_amount": -1}  
15         }  
16     ],  
17     "description": "The ORG directive is used to set the starting memory  
address of the program."  
18 }
```

The JSON data entry for the @ORG directive. This directive takes a numeric argument and is used to set the starting memory address of the program.

The @ORG directive's JSON object has a similar syntax to the @EQU entry we just viewed. It takes a number as an argument. There isn't too much to talk about here, so we'll move on to the next section where we'll look into the program directives.

## Program Directives: @TTL, @SUBT

In this section we'll take a look at the two program directives, @TTL and @SUBT. Both of these directives are used to provide information about the program. Note that the directive's string argument cannot have white space in it. Let's see how these directives are used in an assembly source code file.

*Listing 6-13: Assembly Directive Source Example - @TTL, @SUBT*

```
1 @TTL |AllOpCodeTests|
2 @SUBT |SubTitleTest|
```

A snippet of assembly source code demonstrating the use of the @TTL and @SUBT directives.

Both program directives should be used at the top of an assembly source code file before any area definitions.

*Listing 6-14: Assembly Directive Listing Example - @TTL, @SUBT*

```
1 0000000000 0000000000 ... @TTL |AllOpCodeTests|
2 0000000001 0000000000 ... @SUBT |SubTitleTest|
```

A snippet of assembly listing code demonstrating the use of the @TTL and @SUBT directives indicating no memory address or binary representation.

For the most part these directives are used to identify the program and to provide assembler log entries that correspond to the @TTL and @SUBT directives string arguments. Notice that the directive lines do not have memory address or binary representation information. These directives are for the assembler only, not the program's binary output. A listing showing the GenAsm's standard output in which the current program's title and subtitle are shown.

*Listing 6-15: Assembler Output Example - @TTL, @SUBT*

```
1 Assembler Meta Data:  
2 Title: AssemblerChecks  
3 SubTitle: SubTitleTest  
4 LineLengthBytes: 2  
5 LineLengthWords: 0  
6 LineLengthHalfWords: 1
```

An excerpt from the GenAsm assembler's text output showing the current program's title and subtitle values.

The title and subtitle directives should be used in every program. At the very least the @TTL directive is required. Use them to identify your program. The title and subtitle are present in the assembler's standard output as well as the generated listing files. Using these directives has an inherent benefit that makes identifying the program associated with certain files easier. In the next section we'll take a look at the first group of area directives.

## Area Directives Part 1: @AREA, @ENTRY, @END

The area directives, part 1, are a set of directives used in the definition of a code or data area, as we'll soon see. By definition, directives provide structure to an assembly program, among other things. Part of the structure imposed on a GenAsm assembly program is that it consists of at most two, and at least one area. Let's take a look at the use of the directives necessary for the definition of an area.

*Listing 6-16: Assembly Directive Source Example - @AREA, @ENTRY, @END*

```
1 @AREA |Program| @CODE, @READONLY  
2 @ENTRY  
3 ...  
4 @END
```

An assembly source code listing showing the use of the @AREA, @ENTRY, and @END.

The area definition is quite simple even though it involves a few different directives. For our purposes we'll focus on the @AREA, @ENTRY, and @END directives. The @AREA directive is used to start the definition of an "area" of assembly code or assembly data or both. The string, a DirectiveString argument type, is used to name the title. The remaining directives on this line are used to describe in more detail attributes of this area. We'll cover these directives in the next section of this chapter. For now, we'll just ignore them. The @ENTRY and @END directives are used to wrap the assembly source code for this area and create a segment of contiguous code.

Because the area directive is used to format and structure the assembly code it doesn't end up in the final binary representation of the program. As such we won't review any listing files with regard to these directives. In the next section we'll wrap up the review of the assembler's directives with part 2 of the area directives.

## **Area Directives Part 2: @CODE, @DATA, @READONLY, @READWRITE**

In part 2 of the area directives, we'll take a look at the directives used to define attributes of an assembly code area. In general, there are two types of areas supported by the GenAsm assembler.

*Listing 6-17: Assembly Directive Source Example - @CODE, @DATA, @READWRITE, @READONLY*

```
1 @AREA |Data| @DATA, @READWRITE  
2 @ENTRY  
3 ...  
4 @END
```

An assembly source code listing showing the use of the @DATA, @READWRITE, and @READONLY. Note that the @CODE directive can be seen in use in listing 6-15.

I should mention that the @CODE and @DATA directives are not currently enforced in any way. I do, however, recommend setting up your program to properly separate assembly source code and assembly data entries.

Furthermore, although they are currently defined and supported in syntax the @READWRITE and @READONLY directives are not enforced. I would recommend using them to accurately label your intended use of a given assembly code area. Similar to the directives in part 1 of the area directive review section, the directives in this section do not end up in the final binary representation of the program. They are for the assembler only and are used to organize a program's assembly source code.

## Chapter Conclusion

In this chapter we covered all of the assembler's supported directives. There was a fair amount of material for us to cover and we didn't check the JSON definition of each and every directive due to redundancy but I recommend taking a moment to look over their JSON definition file. Let's summarize the directives we've covered here in the following list.

### //2-BYTE DATA DIRECTIVES

- **@DCHW:** A two-byte data directive used to store data to a memory location in a program.
- **@FLPDCHW:** A two-byte, flipped endian, directive used to store data to a memory location in a program.

### //1-BYTE DATA DIRECTIVES

- **@DCWBF:** A one-byte data directive used to store a single byte of data as a padded two-byte value to a memory location in a program. This directive uses the first byte of the specified value.
- **@DCWBS:** A one-byte data directive used to store a single byte of data as a padded two-byte value to a memory location in a program. This directive uses the second byte of the specified value.
- **@DCB:** A one-byte data directive used to store a single byte of data as a padded two-byte value to a memory location in a program.

### //LABEL AND ADDRESS DIRECTIVES

- **@EQU:** A label directive used to create a value-based label in an assembly program.
- **@ORG:** A memory directive used to set the starting memory address of the current program.

//PROGRAM DIRECTIVES

- **@TTL:** A program directive used to give the current program a title.
- **@SUBT:** A program directive used to give the current program a subtitle.

//AREA DIRECTIVES

- **@ENTRY:** An area directive that marks the start of an assembly source code section.
- **@END:** An area directive that marks the end of an assembly source code section.
- **@AREA:** An area definition directive.
- **@READONLY:** An area definition attribute directive that indicates the area should be read-only.
- **@READWRITE:** An area definition attribute directive that indicates the area should be read-write.
- **@CODE:** An area definition attribute directive that indicates the area should contain code.
- **@DATA:** An area definition attribute directive that indicates the area should contain data.

That brings us to the conclusion of the GenAsm assembler's directive review. Pay special attention to the directives reviewed here as they are crucial to any assembly program you'll write with the GenAsm program. In the

next section we'll review the assembler's directory structure including the expected location of the JSON data files that are used to define the ARM Thumb-1 instruction set.

# **Chapter 7: Instruction Set Data Files - ARM Thumb-1**

We've covered the ARM Thumb-1 instruction set and the supported preprocessor and standard directives. This has given us a lot of information to process. Don't be too concerned if you don't have everything down firmly, you'll get more experience working with the afore mentioned concepts and information as we proceed. You may not be actively aware of it, but we've covered a fair amount of the syntax and grammar involved with our target instruction set. If you think back to our opcode review, Chapters 3 and 4, we defined a syntax of how we want to refer to each opcode and we also, inadvertently, defined what a valid instruction line looks like. In this way we have syntax and grammar, albeit simple grammar.

The way our assembler is designed to work is by loading up a series of JSON data files that describe the opcodes, directives, registers, and valid lines involved with a target instruction set. In our case ARM Thumb-1 instruction set. In the subsequent sections we'll review the data files, including the objects and sub-objects, used to define the instruction set.

Take a moment to think about the bigger picture. Our assembler, as we've defined it, loads up a JSON based data representation of an instruction set and then parses a file containing assembly source code in the matching

syntax and grammar defined by the JSON data files. The source code is pre-processed, lexized, tokenized, resulting in a data structure of known tokens. This data structure is compared to known valid lines of assembly source code as defined by the JSON data files.

If everything works out accordingly there will be a one-to-one match between the assembly source code and the known valid instructions of the given instruction set. In this case, we can move forward and generate listing and binary executable output files. Now, we've seen a fair amount of these JSON objects during our opcode and standard directive review but we're going to go into a bit more detail and make sure we discuss all attributes and sub-objects used to define an entry in a particular JSON data file.

## JSON Data Files: Instruction Sets: `is_sets.json`

The main entry point into the assembler's JSON data files is the `is_sets.json` file. This file is the only file in an instruction set's JSON data files that must have a strict syntax and structure. This file is used to define the data files associated with a given instruction set. In our case we'll be defining our version of the ARM Thumb-1 instruction set. Let's take a look at the structure of the `is_set.json` file.

*Listing 7-1: JSON Data File: `is_sets.json`*

```
01 {
02   "obj_name": "is_sets",
03   "is_sets": [
04     {
05       "obj_name": "is_set",
06       "set_name": "THUMB_ARM7TDMI",
07       "is_files": [
08         ...
09       ]
10     }
11   ]
12 }
```

A snippet of the `is_sets.json` data file showing the main attributes and a skeleton of an `is_set` object with cleared file objects.

The first attribute of the `is_sets` object is the string `obj_name`. This is an attribute that we've seen before in previous JSON object reviews. This attribute is used to define a name for the JSON object. You'll see that in all JSON data files the main objects will have this attribute populated, sub-objects will not. The only other attribute of the `is_sets` object is the `is_sets` attribute which holds an array of entries. Each entry in the array is expected to be an `is_set` object.

You can see in the previous listing that the `is_set` object has the `obj_name` attribute with the proper value set. The next attribute listed, `set_name`, is used to give a name to the instruction set being defined. In this case we're working on the "THUMB\_ARM7TDMI" instruction set. This is just a fancy name for the ARM Thumb-1 set we've been working with. The last attribute of this object is an array of `is_file` entries named `is_files`.

Notice that the list of the files needed to define an instruction set is not defined in any way. It's up to you and the requirements of the instruction set you're working with to come up with a way to define the syntax and grammar of your assembly programming language. Of course, you're provided with an example of a complete implementation of the ARM Thumb-1 instruction set. You can use this as the basis for your own implementation or simply as a basis for learning and exploration. Let us now take a look at the list of files needed to define the Thumb-1 instruction set.

*Listing 7-2: ARM Thumb-1 Instruction Set's JSON Data File List*

```
01 {
02   "obj_name": "is_file",
03   "path": "./cfg/THUMB/is_arg_types.json",
04   "loader_class":
"net.middlemind.GenAsm.Loaders.Thumb.LoaderIsArgTypes",
05   "target_class":
"net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsArgTypes",
06   "category": "arguments"
07 },
08 {
09   "obj_name": "is_file",
10   "path": "./cfg/THUMB/is_entry_types.json",
```

```
11  "loader_class":  
"net.middlemind.GenAsm.Loaders.Thumb.LoaderIsEntryTypes",  
12  "target_class":  
"net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsEntryTypes",  
13  "category": "entries"  
14 },  
15 {  
16  "obj_name": "is_file",  
17  "path": "./cfg/THUMB/is_op_codes.json",  
18  "loader_class":  
"net.middlemind.GenAsm.Loaders.Thumb.LoaderIsOpCodes",  
19  "target_class":  
"net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsOpCodes",  
20  "category": "opcodes"  
21 },  
22 {  
23  "obj_name": "is_file",  
24  "path": "./cfg/THUMB/is_registers.json",  
25  "loader_class":  
"net.middlemind.GenAsm.Loaders.Thumb.LoaderIsRegisters",  
26  "target_class":  
"net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsRegisters",  
27  "category": "registers"  
28 },  
29 {  
30  "obj_name": "is_file",  
31  "path": "./cfg/THUMB/is_valid_lines.json",  
32  "loader_class":  
"net.middlemind.GenAsm.Loaders.Thumb.LoaderIsValidLines",  
33  "target_class":  
"net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsValidLines",  
34  "category": "lines"  
35 },  
36 {  
37  "obj_name": "is_file",  
38  "path": "./cfg/THUMB/is_directives.json",  
39  "loader_class":  
"net.middlemind.GenAsm.Loaders.Thumb.LoaderIsDirectives",  
40  "target_class":  
"net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsDirectives",  
41  "category": "directives"  
42 },  
43 {
```

```
44  "obj_name": "is_file",
45  "path": "./cfg/THUMB/is_empty_data_lines.json",
46  "loader_class":
"net.middlemind.GenAsm.Loaders.Thumb.LoaderIsEmptyDataLines",
47  "target_class":
"net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsEmptyDataLines",
48  "category": "special"
49 }
```

The contents of the `is_files` array for the ARM Thumb-1 instruction set definition contained in the essential data file `is_sets.json`.

Notice that the `obj_name` value for these objects is “`is_file`”, also note that the attribute that these entries are members of is the `is_files` attribute. This is a convention that we’ll see over and over again in the data files. The next attribute of the `is_file` object is the `path`. The `path` attribute is a string value that is a relative path to the JSON data file associated with this entry. Again, this relative path is based on the project’s working directory.

The next two object attributes are very important they describe how to parse and load the data in the associated JSON data file and what Java class to store it in, respectively. The `loader_class` attribute holds the full package and class name of the Java class designed to parse and load this data file. Notice that the loader classes are in the “`Loaders.Thumb`” package.

The target class, used to hold the parsed JSON data, is specified by the `target_class` attribute. Similar to the previous attribute we reviewed, the `target_class` attribute holds the full package and class name of the Java class designed to hold the parsed data. Last but not least we have the `category` attribute. This attribute is used to describe the file entry and give it some added uniqueness.

The main take away here is that the `is_set` object can have as many files as it needs to properly define the instruction set being implemented. What is more, the Java class used to parse the data and the class used to hold it are all configurable from this JSON data file. This gives us an extreme amount of power when it comes to extending the assembler’s capabilities.

Remember, this is the only data file that has to be in a set, known location, and has to have a set, known, structure. The reason for this is that because it is static, we can use it as a foundation for supporting our assembler's extensibility. In the next section we'll take a look at the first JSON data file to review.

## JSON Data Files: Argument Types: `is_arg_types.json`

The `is_arg_types.json` data file is used to hold information about the instruction set's argument types and how they map to entry types that can be provided to opcodes and/or directives. This is one of the simpler JSON data files, so we'll review it quickly.

*Listing 7-3: JSON Data File: `is_arg_types.json`*

```
1 {  
2     "obj_name": "is_arg_types",  
3     "set_name": "THUMB_ARM7TDMI",  
4     "is_arg_types": [  
5         ...  
6     ]  
7 }
```

A snippet of the `is_arg_types.json` data file showing the main attributes with cleared arg type objects.

The `obj_name` attribute is set to a value of “`is_arg_types`” to indicate the name of the object. The next attribute of this object, `set_name`, holds the name of the instruction set represented. In this case we’re working on the “`THUMB_ARM7TDMI`” instruction set. This is the ARM Thumb-1 instruction set we reviewed earlier; we’ve just given it a more complete name here. The last attribute of the object shown is the `is_arg_types` attribute. It’s used to hold a list of `is_arg_type` objects. Let’s take a look at what they look like.

*Listing 7-4: Example Entries for the `is_arg_types.json` Data File*

```
01 {  
02     "obj_name": "is_arg_type",
```

```

03     "arg_name": "Group",
04     "is_entry_types": ["GroupStart", "GroupStop"]
05 },
06 {
07     "obj_name": "is_arg_type",
08     "arg_name": "Label",
09     "is_entry_types": ["Label"]
10 },
11 {
12     "obj_name": "is_arg_type",
13     "arg_name": "LabelRef",
14     "is_entry_types": ["LabelRef"]
15 },
16 ...
17 {
18     "obj_name": "is_arg_type",
19     "arg_name": "SWord7",
20     "is_entry_types": ["Number"]
21 },
22 {
23     "obj_name": "is_arg_type",
24     "arg_name": "Word8",
25     "is_entry_types": ["Number"]
26 },
27 {
28     "obj_name": "is_arg_type",
29     "arg_name": "Value8",
30     "is_entry_types": ["Number"]
31 },
32 {
33     "obj_name": "is_arg_type",
34     "arg_name": "DirectiveString",
35     "is_entry_types": ["String"]
36 }

```

The abridged contents of the `is_arg_types` array showing a few example `is_arg_type` object entries.

In the previous listing we have an abridged array of instruction set argument type objects. These object entries connect instruction set argument

data types to the assembler's entry types, `is_entry_type` objects. In other words, this is a mapping between instruction set argument types, those defined by the instruction set documentation, and the assembler's entry types, the data types used by the GenAsm assembler.

For instance, the `Value8` argument type, you may recall this from our opcode review, is mapped to a `Number` entry type. This means that the assembler views the argument data as numeric. There are many different argument data types that can be represented as numeric. The `Word8` and `SWord7` argument types are also considered numeric. In this way the mapping between the assembler's entry types and the instruction set's argument types creates an abstraction layer that allows us to handle opcode arguments more effectively by associating similar types of data.

Note that the `is_arg_types` JSON data file doesn't define any pattern matching information. The file only presents mapping data between two different types of arguments, the instruction set's and the assembler's. In the next section we'll take a look at the `is_entry_types` JSON data file that defines the assembler's entry types and describes how to extract them from the assembly source text.

## JSON Data Files: Entry Types: `is_entry_types.json`

The `is_entry_types.json` data file is used to hold information about the entry types that can be provided to opcodes and/or directives, the opcodes and directives themselves, and how to extract them from assembly source code. This is a bit different than the JSON data file we looked at in the previous section. The data contained in the `is_entry_types.json` data file defines aspects of the assembler's syntax.

The `is_arg_types.json` file did not provide any syntax data. Instead, the file maintains a mapping between higher level, assembler, data types and lower level, instruction set, data types. These data types manifest themselves as arguments to opcodes and/or directives. Because this data file has information on how to extract "entries" from assembly source code i.e., convert text artifacts to known token objects. It defines the syntax of the assembler.

Grammar, semantics, and syntax are key aspects of any programming language, including assembly language. While the role of grammar in assembly programming is reduced due to the simplicity of the assembly programming language's linear structure, it's still there. We'll get into the grammar of assembly language in just a bit when we review the valid lines JSON data file. For now, let's focus on the entry types and how they define the assembler's syntax.

*Listing 7-5: JSON Data File: is\_entry\_types.json*

```
1 {
2     "obj_name": "is_entry_types",
3     "set_name": "THUMB_ARM7TDMI",
4     "is_entry_types": [
5         ...
6     ]
7 }
```

A snippet of the `is_entry_types.json` data file showing the main attributes with cleared entry type objects.

As we've seen before, and as we'll soon see again, this JSON object has the `obj_name` attribute. This attribute is used to give a proper object name to the JSON entry. The `set_name` attribute is used to define which instruction set this data file is associated with. Lastly, the `is_entry_types` attribute holds an array of `is_entry_type` objects. Let's take a look at a sample of the file's `is_entry_type` objects.

*Listing 7-6: Example Entries for the is\_entry\_types.json Data File*

```
001 {
002     "obj_name": "is_entry_type",
003     "type_name": "Comment",
004     "type_category": "Comment",
005     "category": "Comment",
006     "category_class": "java.lang.String",
007     "txt_match": {
008         "starts_with": [";"],
009         "contains": ["*"],
```

```

010     "must_contain": [],
011     "must_not_contain": [],
012     "ends_with": ["*"]
013   }
014 },
015 {
016   "obj_name": "is_entry_type",
017   "type_name": "RegisterRangeLow",
018   "type_category": "RegisterLow",
019   "category": "Arg",
020   "category_class":
"net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsRegister",
021   "txt_match": {
022     "starts_with": ["R"],
023     "contains": ["0~7", "-", "R"],
024     "must_contain": ["-"],
025     "must_not_contain": ["R8", "R9", "R10", "R11", "R12", "R13",
"R14", "R15"],
026     "ends_with": [",", ";", "0~7"]
027   }
028 },
029 {
030   "obj_name": "is_entry_type",
031   "type_name": "RegisterLow",
032   "type_category": "RegisterLow",
033   "category": "Arg",
034   "category_class":
"net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsRegister",
035   "txt_match": {
036     "starts_with": ["R"],
037     "contains": ["0~7"],
038     "must_contain": [],
039     "must_not_contain": ["R8", "R9", "R10", "R11", "R12", "R13",
"R14", "R15"],
040     "ends_with": [",", ";", "0~7"]
041   }
042 },
043 {
044   "obj_name": "is_entry_type",
045   "type_name": "Number",
046   "type_category": "Number",
047   "category": "Arg",
048   "category_class": "java.lang.Double",

```

```

049     "txt_match": {
050         "starts_with": ["#bx", "#-", "#", "0~9"],
051         "contains": ["0~9"],
052         "must_contain": [],
053         "must_not_contain": [],
054         "ends_with": ["0~9", ",", ";"]
055     }
056 },
057 {
058     "obj_name": "is_entry_type",
059     "type_name": "Label",
060     "type_category": "Label",
061     "category": "Label",
062     "category_class": "java.lang.String",
063     "txt_match": {
064         "starts_with": ["a~z"],
065         "contains": ["a~z", "a~z0~9", "_"],
066         "must_contain": [],
067         "must_not_contain": [],
068         "ends_with": ["a~z0~9", "a~z", "_", ";", ","]
069     }
070 },
071 {
072     "obj_name": "is_entry_type",
073     "type_name": "LabelRef",
074     "type_category": "LabelReference",
075     "category": "Arg",
076     "category_class": "java.lang.String",
077     "txt_match": {
078         "starts_with": [=, ~, -, `],
079         "contains": ["a~z", "a~z0~9", "_"],
080         "must_contain": [],
081         "must_not_contain": [],
082         "ends_with": ["a~z0~9", "a~z", "_", ";", ","]
083     }
084 },
085 {
086     "obj_name": "is_entry_type",
087     "type_name": "OpCode",
088     "type_category": "OpCode",
089     "category": "OpCode",
090     "category_class": "net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsOpCode",

```

```

091     "txt_match": {
092         "starts_with": ["A~Z"],
093         "contains": ["A~Z"],
094         "must_contain": [],
095         "must_not_contain": [],
096         "ends_with": ["A~Z"]
097     }
098 },
099 {
100     "obj_name": "is_entry_type",
101     "type_name": "Directive",
102     "type_category": "Directive",
103     "category": "Directive",
104     "category_class":
"net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsDirective",
105     "txt_match": {
106         "starts_with": ["@"],
107         "contains": ["A~Z"],
108         "must_contain": ["@"],
109         "must_not_contain": [],
110         "ends_with": ["A~Z", " "]
111     }
112 },
113 {
114     "obj_name": "is_entry_type",
115     "type_name": "DirectiveString",
116     "type_category": "String",
117     "category": "DirectiveArg",
118     "category_class": "java.lang.String",
119     "txt_match": {
120         "starts_with": ["|"],
121         "contains": ["a~z", "A~Z"],
122         "must_contain": ["|"],
123         "must_not_contain": [],
124         "ends_with": ["|"]
125     }
126 }
```

The abridged contents of the `is_entry_types` array showing a few example `is_entry_type` objects.

The previous listing shows an abridged version of the JSON data file included with the GenAsm assembler project. Note that a number of redundant entries have been removed. Please make sure you look over the complete file located at the following relative path.

```
.cfg\THUMB\is_entry_types.json
```

Now that you've had a minute to look over the data entries, does anything stand out to you? Well one thing that should come to mind is that there are multiple entries in the file that map to the same `type_name` attribute. For instance, there are multiple entries that map to a type of `Number` because there are many ways to define a number value in an assembly source file. Think about this for a moment, we've encountered this concept before during the review of the argument types JSON data file.

Let's take a moment to talk about the attributes of an `is_entry_type` objects shown in the previous listing. The `obj_name` attribute we've seen before. It provides a proper name for the JSON object. The next four attributes are descriptive and used to set type and category information for the object. The `type_name` attribute holds the name of this entry type object.

The `name` attribute is used to reference this entry type object in other JSON data files. If you look back to the `is_arg_types.json` section, you'll see it in use. Note that we use an array attribute when storing entry types in most situations. That's because it's often the case that multiple entry types can be applied to the current attribute.

The next attribute, `type_category`, is a generic attribute used to provide a higher-level grouping of the entry type. You can see in listing 7-6 that there are multiple ways to define a `RegisterLow` category of entry types. This makes sense if you think about it for a moment. If you define a low register directly or through a register range expression, you're still defining a low register. For this reason, the type of category is the same for both entries.

The following attribute, `category`, holds a second piece of general descriptive data. The `category` field should be used to group entry types by their place in the assembly source grammar. An example of this is the Arg,

DirectiveArg, OpCode, Directive, Label, and Comment category values. Subsequently, the `category_class` attribute holds the full package name of the Java class that best represents the entry type.

Lastly, we have the `txt_match` attribute which holds a special sub-object that defines how, in text, the entry type is described. It's this data that is used to drive the tokenizer. Recall that the tokenizer is run after the preprocessor and the lexer and it's used to convert text artifacts into tokens. We can start to see what types of tokens the assembler will support.

Let's take a look at the attributes of the text match sub-object. First off, how can you tell that it's a sub-object? Does anything about the object's attributes stand out to you? Well for one thing, it's a valid JSON object. Secondly, it doesn't have the `obj_name` attribute. That indicates to us that this is a sub-object, and as such it has a leaner JSON representation. The text match attributes are five different arrays used to define a unique character pattern used to identify the given entry type. Let's take a quick look at an example, listed subsequently.

*Listing 7-7: Example DirectiveString Text Match Sub-Object*

```
1 {
2   "starts_with": ["|"],
3   "contains": ["a~z", "A~Z"],
4   "must_contain": ["|"],
5   "must_not_contain": [],
6   "ends_with": ["|"]
7 }
```

An example of a text match sub-object specifically of the DirectiveString entry type object.

As you can see from the previous listing, sub-objects are generally smaller and more concise than the other JSON objects we've been reviewing. The first attribute, `starts_with`, is an array and it defines all the possibly starting characters used to define an entry of the given type. In the previous listing the entry has to start with the pipe character, '|'. The following attribute,

`contains`, is also an array and it's used to list all the valid characters that can be contained between the starting and ending characters.

In the provided example the `DirectiveString` entry type can contain lower- and upper-case characters. The lower- and upper-case character entries are an example of a special entry. Special entries can only be used with the following attributes, `starts_with`, `contains`, and `ends_with`. The following special entries are supported.

- **a~z**: All lower-case letters and underscore characters.
- **A~Z**: All upper-case letters and underscore characters.
- **a~z0~9**: All lower-case letters, underscore, and numeric characters.
- **#~#**: A number range i.e., 0~7, of supported numbers.

The next two attributes `must_contain` and `must_not_contain`, are array attributes that take lists of characters. Clearly the entry must contain any characters in the `must_contain` array and must not include any characters in the `must_not_contain` array. Lastly the `ends_with` attribute can be used to define an entry type's ending character. Using this sub-object, you can define the unique set of characters necessary to define an entry of a certain type.

It's important to note that by design there is no overlap between entry types. This ties back to that comment from previous chapters about the assembler being forward looking. Because each entry type is unique the assembler can know the type of the token it's scanning after it's done running the tokenizer. Recall that the tokenizer is used to convert text artifacts to tokens. In the next section we'll take a look at the opcode data file. We've seen quite a bit of this file during the opcode review in Chapters 3 and 4.

## JSON Data Files: OpCodes: `is_opcodes.json`

The next data file for us to review is one of the most important sources of data because it describes the full ARM Thumb-1 instruction set in an array of JSON objects. We've seen a number of the entries in this data file during the opcode review but we're going to take another look at some entries and really dive into the meaning and use of the objects and sub-objects involved.

*Listing 7-8: JSON Data File: is\_opcodes.json*

```
01 {
02     "obj_name": "is_op_codes",
03     "set_name": "THUMB_ARM7TDMI",
04     "endian": "BIG",
05     "pc_prefetch_bits": 32,
06     "pc_prefetch_bytes": 4,
07     "pc_prefetch_halfwords": 2,
08     "pc_prefetch_words": 1,
09     "pc_alignment": "WORD",
10     "pc_lsb_zeroed": true,
11     "nop_assembly": "MOV R8, R8 ;nop instruction",
12     "nop_binary": "0100011011000000",
13     "nop_hex": "46C0",
14     "bit_series": {"bit_start": 0, "bit_stop": 15, "bit_len": 16},
15     "is_op_codes": [
16         ...
17     ]
18 }
```

A snippet of the `is_opcodes.json` data file showing the main attributes with the opcode entries removed.

The first attribute is a familiar one, `obj_name`, it holds the proper name of the JSON object. The second attribute should also look familiar, the `set_name` attribute. This holds the name of the set that the data file belongs to. I should mention that the data files, as described thus far, are unique to our implementation of the ARM Thumb-1 instruction set. If you are working on your own instruction set definition you may find that you need different files, more files, or adjusted object/attributes.

That's perfectly fine. Our implementation is not meant to be a template for all other implementations, it's meant to be an example. The next attribute, `endian`, is used to represent the endianness used by the opcode documentation. The next four attributes are all ways of describing the prefetch of the target instruction set. The `pc_alignment` attribute is a string value used to represent the default alignment of this instruction set. The value used here is actually a bit misleading.

Some of the opcodes require WORD alignment of a value's address, while others are HALFWORD aligned. I set the value for this attribute to WORD because word alignment is also half-word alignment. In any case, the value of this attribute is more descriptive than anything else. Following this is the `pc_lsb_zeroed` attribute which indicates that the least significant bit is zeroed. This indicates that half-word alignment is required.

The subsequent three fields – `nop_assembly`, `nop_binary`, `nop_hex` – are all used to define a no-operation instruction for this instruction set. The `nop_binary` and `nop_hex` attributes are used to hold the actual binary and hexadecimal representations of the non-operation instruction. Notice that the actual assembly source code for the non-operation instruction copies a register onto itself essentially doing nothing.

I should note that the preprocessor uses the same assembly source code for the `$NOP` preprocessor directive. However, this data is not loaded from this JSON data file. The preprocessor runs well before the JSON data files are loaded and parsed. The non-operation instruction is not data driven by this attribute. The next attribute is the `bit_series` attribute and it holds a sub-object that describes the binary series of one instruction.

Let's take a look at this sub-object in detail. Note that it doesn't define any values in the binary sequence, but it does describe the start bit, the stop bit, and the length of the bit series. In the example listed previously the `bit_series` shows a 16-bit instruction with starting bit set to 0 and the ending bit set to 15. The last attribute we have to discuss is the `is_op_codes` array. This attribute holds an array of opcode definition objects. In order to make sure we have a good understanding of how the opcodes are defined I want to spend some time reviewing them. Let's take a look at one of the more complex opcode object entries.

*Listing 7-9: JSON Data File: `is_opcodes.json` Example `is_op_code` Entry*

```
01 {  
02   "obj_name": "is_op_code",  
03   "op_code_name": "LDR",  
04   "index": 38,  
05   "arg_len": 3,  
06   "arg_separator": ",",
```

```

07     "bit_rep": {"bit_string": "01001", "bit_int": 9, "bit_len": 5},
08     "bit_series": {"bit_start": 11, "bit_stop": 15, "bit_len": 5},
09     "args": [
10         {
11             "obj_name": "is_op_code_arg",
12             "arg_index": 0,
13             "bit_index": 1,
14             "is_entry_types": ["RegisterLow"],
15             "is_arg_type": "Rd",
16             "bit_series": {"bit_start": 8, "bit_stop": 10, "bit_len": 3}
17         },
18         {
19             "obj_name": "is_op_code_arg",
20             "arg_index": 1,
21             "bit_index": 0,
22             "is_entry_types": ["GroupStart"],
23             "is_arg_type": "Group",
24             "bit_series": {"bit_start": 0, "bit_stop": 7, "bit_len": 8},
25             "sub_arg_separator": ",",
26             "sub_args": [
27                 {
28                     "obj_name": "is_op_code_arg",
29                     "arg_index": 0,
30                     "bit_index": -1,
31                     "is_entry_types": ["RegisterPc"],
32                     "is_arg_type": "PC",
33                     "bit_series": {"bit_start": -1, "bit_stop": -1,
34 "bit_len": -1}
35                 },
36                 {
37                     "obj_name": "is_op_code_arg",
38                     "arg_index": 1,
39                     "bit_index": 0,
40                     "is_entry_types": ["Number", "LabelRef"],
41                     "is_arg_type": "Word8",
42                     "bit_series": {"bit_start": 0, "bit_stop": 7,
43 "bit_len": 8},
44                     "num_range": {"min_value": 0, "max_value": 1020,
45 "bit_len": 10, "twos_complement": false, "alignment": "WORD"},
46                     "bit_shift": {"shift_dir": "RIGHT", "shift_amount": 2}
47                 }
48             ]
49         }
50     }
51 
```

```
47      ]
48 }
```

An example is\_op\_code object entry of fair complexity.

The first attribute in the previously listed example is, as expected, the `obj_name` attribute. The value of `obj_name` for this object is “`is_op_code`”. The following attribute, `op_code_name`, holds the name of the current instruction. This value is not unique, some instructions are defined multiple times with different argument signatures. The `index` attribute, line 4, holds the index of the current opcode in the array of opcodes.

The `arg_len` attribute holds an integer value indicating the number of arguments for this instruction at the top-level. Groups and lists are viewed as single top-level arguments so with regard to the example listing the opcode has three arguments. The next attribute, `arg_separator`, holds a string value that indicates the character used to separate the opcode’s arguments.

Next up is the `bit_rep` attribute. This attribute holds a sub-object designed to represent this opcode’s binary signature. This is the first time we’ve seen encoded binary signatures in the JSON data files. Since this information is part of the opcode documentation, we get it for free and include it as part of the opcode definition. Let’s take a look at an example of the sub-object.

*Listing 7-10: Example LDR OpCode’s Bit Representation Sub-Object*

```
1 {
2   "bit_string": "01001",
3   "bit_int": 9,
4   "bit_len": 5
5 }
```

An example of a bit representation sub-object.

The attributes of the bit representation sub-object start with the `bit_string`, a string holding a binary number. The next attribute, `bit_int`,

is an integer representation of the binary string. Lastly, the `bit_len` attribute describes the length of the binary string. Return your attention back to the opcode entry object, line 8. The `bit_series` attribute holds a sub-object that describes the placement of the bit representation data in the final binary encoding of the instruction.

The assembler assumes that the initial binary representation of each line of assembly source code is big-endian and that an endian flip will result in a little-endian encoding. Keep this in mind when you're working on the example programs and your own programs. Let's look at the sub-object up close.

*Listing 7-11: Example LDR OpCode's Bit Series Sub-Object*

```
1 {  
2   "bit_start": 11,  
3   "bit_stop": 15,  
4   "bit_len": 5  
5 }
```

An example of a bit series sub-object.

The sub-object has three attributes shown previously. The first entry, `bit_start`, is an integer and it represents the first bit position in a bit series. The `bit_stop` attribute is also an integer value and it represents the last bit in a bit series. The last attribute, `bit_len`, is an integer value that represents the total length of the bit series. The bit series sub-object is used to indicate the bit series, bit position, of information in the resulting binary encoding for a given line of assembly source code.

With regard to the main `is_op_codes` object the bit series sub-object indicates where, in the final binary representation, for the given opcode, that the bit representation object data is placed. The next attribute `args` holds an array of `is_op_code_arg` JSON objects. Each opcode argument entry represents an argument for the given opcode.

*Listing 7-12: Example Simple is\_op\_code\_arg Object*

```
1 {
2   "obj_name": "is_op_code_arg",
3   "arg_index": 0,
4   "bit_index": 1,
5   "is_entry_types": ["RegisterLow"],
6   "is_arg_type": "Rd",
7   "bit_series": {"bit_start": 8, "bit_stop": 10, "bit_len": 3}
8 }
```

An example `is_op_code_arg` object that represents a register-based opcode argument.

The `is_op_code_arg` object is an important part of an opcode definition so we'll review each attribute in detail. The `obj_name` attribute is to be expected and has a value of “`is_op_code_arg`”. The `arg_index` attribute is an integer value that indicates this argument's position in the assembly source code for the associated opcode. The `bit_index` attribute, line 4, is similar except that it indicates the argument's position in the binary encoding of the current instruction.

The `is_entry_types` attribute is an array of strings of allowed, associated `is_entry_type` names. In many cases, though not in the previously listed one, there are multiple entry types that can satisfy an argument's requirements. Next, the `is_arg_type` attribute is a name for the argument. In most cases the string will match the argument's name in the instruction set documentation. The last attribute to discuss is the `bit_series` entry. This sub-object represents the argument's position in the binary representation of the instruction.

That concludes our review of the more basic opcode argument JSON object. The way the JSON objects are setup they are often overloaded to support describing different variations of opcode arguments. In this case there are a few different opcode argument types. We just finished looking at the more basic register type, next we'll look at a more complicated number argument type.

*Listing 7-13: Example Complex `is_op_code_arg` Object 1*

```

01 {
02   "obj_name": "is_op_code_arg",
03   "arg_index": 1,
04   "bit_index": 0,
05   "is_entry_types": ["GroupStart"],
06   "is_arg_type": "Group",
07   "bit_series": {"bit_start": 0, "bit_stop": 7, "bit_len": 8},
08   "sub_arg_separator": ",",
09   "sub_args": [
10     ...
11   ]
12 }

```

An example `is_op_code_arg` object that represents a group start opcode argument.

The next `is_op_code_arg` example for us to look at, listed previously, is a bit more complicated. This is an example of an argument that has its own argument set called sub-arguments. Let's review this permutation of the opcode argument object's new attributes in detail. The first new attribute listed is the `sub_arg_separator` attribute. The value here represents the character used to separate the group's arguments. This value doesn't data drive the code. One potential upgrade to this implementation is to add a data driven argument separator.

Since we are only working with the ARM THUMB-1 instruction set the attribute is included for symmetry with the parent object and to provide documentation. The next entry is the `sub_args` attribute. This attribute is another array of `is_op_code_arg` objects. They follow the same format as many of the previous examples we've already seen so we won't review it again here.

*Listing 7-14: Example Complex `is_op_code_arg` Object 2*

```

01 {
02   "obj_name": "is_op_code_arg",
03   "arg_index": 1,
04   "bit_index": 0,
05   "is_entry_types": ["Number", "LabelRef"],
06   "is_arg_type": "Word8",

```

```

07      "bit_series": {"bit_start": 0, "bit_stop": 7, "bit_len": 8},
08      "num_range": {"min_value": 0, "max_value": 1020, "bit_len": 10,
"twos_complement": false, "alignment": "WORD"},
09      "bit_shift": {"shift_dir": "RIGHT", "shift_amount": 2}
10 }

```

An example is\_op\_code\_arg object that represents a numeric opcode argument.

The last example we'll look at is the permutation used for numbers. Notice that the `is_entry_type` attribute has two entries because this argument can be satisfied by a hard coded numeric value, a number written into the assembly source code, or a label reference value, a number associated with a label or a label's address. The assembler syntax, as implemented for the ARM Thumb-1 instruction set, supports getting different values from a label reference. The supported label reference values are as follows.

- **Label Reference Address =:** Used to return the address of the label referenced.
- **Label Reference Value ~:** Used to return the value, if available, of the label referenced address.
- **Label Reference Offset Address -:** Used to return the offset to the address of the label referenced taking into account the prefetch offset.
- **Label Reference Offset Address Less Prefetch `:** Used to return the offset to the address of the label referenced ignoring the prefetch offset.

There are two new sub-objects listed in the example that we'll review in detail. The first is the number range sub-object stored in the argument's `num_range` attribute. Let's take a look at this object in detail.

*Listing 7-15: Example Number Range Sub-Object*

```

1 {
2     "min_value": 0,

```

```
3     "max_value": 1020,  
4     "bit_len": 10,  
5     "twos_complement": false,  
6     "alignment": "WORD"  
7 }
```

An example of a number range sub-object.

The number range sub-object is used to describe the valid range a numeric argument can have. Because some opcode arguments take values that have a limited number of bits different arguments will have different valid value ranges. Let's review the sub-object's attributes. The `min_value` attribute specifies the minimum value of the range. The `max_value` attribute specifies the maximum value of the range. The bit length argument specifies the length of the binary representation of the number prior to bit shifting.

There are two number encoding flags, `ones_complement` and `twos_complement`. If these flags are set the number's binary representation will undergo the specified binary adjustment. The last entry is the alignment attribute and it's used to indicate the alignment needed for this numeric value. This attribute is used as documentation and does not data drive any assembler code. In the next section we'll take a look at the registers JSON data file.

## JSON Data Files: Registers: `is_registers.json`

The next JSON data file that we'll look at lists all the registers supported by the Thumb-1 instruction set. This information is defined by the `is_registers.json` file. Let's take a look at the main JSON object with entry data removed, listed subsequently.

*Listing 7-16: JSON Data File: `is_registers.json`*

```
1 {  
2     "obj_name": "is_registers",  
3     "set_name": "THUMB_ARM7TDMI",  
4     "is_registers": [  
5         ...
```

```
6      ]
7 }
```

An example of the `is_registers` JSON object with register list entries removed.

The attributes of this JSON object should be familiar, so we won't go over the redundant entries. That brings us to the `is_registers` attribute which holds an array of `is_register` JSON objects. Let's look at what those objects look like next.

*Listing 7-17: JSON Data File: is\_registers.json Example is\_register Objects*

```
01 {
02   "register_name": "R0",
03   "obj_name": "is_register",
04   "is_entry_type": "RegisterLow",
05   "bit_rep": {"bit_string": "000", "bit_int": 0, "bit_len": 3},
06   "type": "LOW",
07   "desc": "general purpose"
08 },
09 {
10   "register_name": "R1",
11   "obj_name": "is_register",
12   "is_entry_type": "RegisterLow",
13   "bit_rep": {"bit_string": "001", "bit_int": 1, "bit_len": 3},
14   "type": "LOW",
15   "desc": "general purpose"
16 },
17 {
18   "register_name": "R2",
19   "obj_name": "is_register",
20   "is_entry_type": "RegisterLow",
21   "bit_rep": {"bit_string": "010", "bit_int": 2, "bit_len": 3},
22   "type": "LOW",
23   "desc": "general purpose"
24 },
25 {
26   "register_name": "R3",
27   "obj_name": "is_register",
28   "is_entry_type": "RegisterLow",
```

```
29     "bit_rep": {"bit_string": "011", "bit_int": 3, "bit_len": 3},
30     "type": "LOW",
31     "desc": "general purpose"
32 }
33 ...
```

An example, unabridged, list of `is_register` objects.

The previously listed example shows an abridged array of `is_register` objects. Each entry represents a valid register or register alias. Please take a moment to look over the data file and make sure you are familiar with all the available registers and any aliases. It's important for us to go into detail so let's take a closer look at the `is_register` object.

*Listing 7-18: Example `is_register` Object*

```
01 {
02     "register_name": "R0",
03     "obj_name": "is_register",
04     "is_entry_type": "RegisterLow",
05     "bit_rep": {"bit_string": "000", "bit_int": 0, "bit_len": 3},
06     "type": "LOW",
07     "desc": "general purpose"
08 }
```

An example `is_register` object from the `is_registers.json` file.

The first object attribute, `register_name`, holds a string representation of the given register. Keep in mind some high registers have aliases and there will be multiple entries in the file with the same `bit_rep` sub-object. The `obj_name` and `is_entry_type` attributes we've seen before. The `obj_name` entry holds the proper name of the JSON object, "is\_register". The `is_entry_type` attribute indicates, by name, which entry type this register is associated with.

The `bit_rep` attribute holds a sub-object that describes the binary representation of the current register. The `type` attribute is a descriptive

attribute that's used to describe the register as high or low. The final attribute, `desc`, is used to provide a general description of the register for documentation purposes. That brings us to the end of this section. In the next section we'll take a look at a data file that helps define the assembly language's grammar.

## JSON Data Files: Valid Source Lines: `is_valid_lines.json`

The next JSON data file for us to look at provides a definition of the assembly language's grammar by defining valid lines of assembly source. Due to the simplicity of the assembly code language, we can get away with this. If we were working on a higher-level programming language this would probably be a bad idea. In our case it works out well because there are only a few valid permutations of different entry types.

Let's step back and think about things from a wider perspective. We've defined opcodes and their arguments. This provides us with a syntax of sorts as we've now defined the way to use an opcode and provide its arguments. With the inclusion of the valid lines data, we now have a definition of what `is_entry_types` can exist on a single line of assembly source code and in what order.

Because all assembly code is based on single lines of instructions, a program constitutes a series of these instructions. We don't encounter complex grammar issues like we could potentially encounter in a higher-level language where recursive syntax, among other things, can exist. Using this to our advantage we can easily define all the valid lines in an assembly program. Let's look at the data file next to demonstrate what we mean.

*Listing 7-19: JSON Data File: `is_valid_lines.json`*

```
1 {
2   "obj_name": "is_valid_lines",
3   "set_name": "THUMB_ARM7TDMI",
4   "min_line_entries": 1,
5   "max_line_entries": 4,
6   "is_valid_lines": [
7     ...
8   ]
```

```
9 }
```

An example of the `is_valid_lines` JSON object with valid line entries removed.

The `is_valid_lines` JSON object looks much like other main JSON objects we've viewed previously. The valid line entry data has been removed from this sample, as in previous examples, for brevity. The first attribute of the object is the `obj_name` attribute which holds a string representation of the object's proper name. The `set_name` attribute is used to indicate which instruction set this data file is associated with.

The next two attributes are used to define the minimum and maximum number of entries allowed on a valid line of assembly source code, lines 4 - 5. The last entry in the example is the `is_valid_lines` attribute which holds an array of valid line objects. Let's take a look at them next.

*Listing 7-20: JSON Data File: `is_valid_lines.json` Example `is_valid_line` Objects*

```
01 {
02     "obj_name": "is_valid_line",
03     "index": 0,
04     "empty_line": true,
05     "is_valid_line": [
06         {"obj_name": "is_valid_line_entry", "is_entry_types": [
07             ["Comment"], "index": 0}
08     ],
09     {
10         "obj_name": "is_valid_line",
11         "index": 1,
12         "empty_line": true,
13         "is_valid_line": [
14             {"obj_name": "is_valid_line_entry", "is_entry_types": ["Label"],
15             "index": 0}
16         ],
17     {
18         "obj_name": "is_valid_line",
19         "index": 2,
```

```

20     "empty_line": true,
21     "is_valid_line": [
22         {"obj_name": "is_valid_line_entry", "is_entry_types": ["Label"], "index": 0},
23         {"obj_name": "is_valid_line_entry", "is_entry_types": ["Comment"], "index": 1}
24     ]
25 },
26 {
27     "obj_name": "is_valid_line",
28     "index": 5,
29     "empty_line": false,
30     "is_valid_line": [
31         {"obj_name": "is_valid_line_entry", "is_entry_types": ["OpCode"], "index": 0},
32         {
33             "obj_name": "is_valid_line_entry",
34             "is_entry_types": ["RegisterRangeHi", "RegisterRangeLow", "RegisterLr", "RegisterPc", "RegisterSp", "RegisterHi", "RegisterLow", "RegisterWb", "Number", "ListStart", "ListStop", "GroupStart", "GroupStop", "LabelRef"],
35             "index": 1
36         }
37     ]
38 },
39 ...

```

An abridged example listing of `is_valid_line` objects.

The `is_valid_line` objects are more consistent than some of the other objects we've seen but, they have variability in that the valid entries defined for a particular valid line are defined in an array attribute, `is_valid_line`. Let's review the attributes of the `is_valid_line` object before looking at the child object, `is_valid_line_entry`, entries. The first attribute is familiar, `obj_name`, and holds a string reference that is the proper name of this JSON object. The next entry, `index`, indicates which position this object has in the array of valid line objects.

The next attribute, `empty_line`, is a Boolean value that indicates if the line of assembly source code will be in the final binary encoding. In this way

an empty line is a line with no opcode, directive or other that ends up in the final binary representation of the program. The final attribute in the object is the `is_valid_line` array which holds a list of `is_valid_line_entry` objects. We'll take a look at a set of them next.

*Listing 7-21: JSON Data File: is\_valid\_lines.json Example is\_valid\_line Objects*

```
01 {  
02     "obj_name": "is_valid_line_entry",  
03     "is_entry_types": ["Label"],  
04     "index": 0  
05 },  
06 {  
07     "obj_name": "is_valid_line_entry",  
08     "is_entry_types": ["Comment"],  
09     "index": 1  
10 }
```

An example list of `is_valid_line_entry` objects.

The `is_valid_line_entry` object is fairly simple. The first attribute is the `obj_name` attribute we know and love. It holds the proper name for the JSON object. The next attribute is the very important `is_entry_types` attribute. This array lists all the valid entry types that can satisfy the current position in the current valid line. The `index` attribute indicates the position of this `is_valid_line_entry` in the current valid line.

The `is_entry_types` attribute holds an array of strings that are the names of an `is_entry_type` object. This attribute lists all the different entry types that can be used to satisfy this position in the valid line. As you can see, the structure of the line is mapped out in the array of valid line entries. Be sure to take a moment and look through this JSON data file to familiarize yourself with the different supported valid lines. In the next section we'll take a look at the directive definitions.

## JSON Data Files: Directives: `is_directives.json`

The `is_directives.json` data file is used to define all the standard directives supported by the assembler. Can you think of a reason why we haven't seen a preprocessor directive JSON data file? If you recall from the preprocessor directive review, there is no data file defining them because they are handled before any data files are loaded.

*Listing 7-22: JSON Data File: `is_directives.json`*

```
1 {
2     "obj_name": "is_directives",
3     "set_name": "THUMB_ARM7TDMI",
4     "is_directives": [
5         ...
6     ]
7 }
```

An example of the `is_directives` JSON object with directive entries removed.

The `is_directives` object is very simple. The attributes `obj_name` and `set_name` we've seen time and time again, so I'll skip reviewing them again here. The `is_directives` attribute is an array that holds all of the directive definitions. The standard directives are a bit different than opcodes in that not every directive is part of the final binary representation of the program. In other words, some directives are used by the assembler only and are not actually part of the resulting program. Next up, let's look at some `is_directive` object examples.

*Listing 7-23: JSON Data File: `is_directives.json` Example `is_directive` Objects*

```
01 {
02     "obj_name": "is_directive",
03     "directive_name": "@ORG",
04     "arg_len": 1,
05     "args": [
06         {
07             "obj_name": "is_directive_arg",
08             "arg_index": 0,
```

```

09      "bit_index": 0,
10      "is_entry_types": ["Number"],
11      "is_arg_type": "Number",
12      "bit_series": {"bit_start": 0, "bit_stop": 31, "bit_len": 32},
13      "num_range": {"min_value": 0, "max_value": 2147483647,
14      "bit_len": 32, "twos_complement": false, "alignment": "WORD"},
15      "bit_shift": {"shift_dir": "NONE", "shift_amount": -1}
16    }
17  ],
18  "description": "The ORG directive is used to set the starting memory
address of the program."
19 },
20  {
21    "obj_name": "is_directive",
22    "directive_name": "@TTL",
23    "arg_len": 1,
24    "args": [
25      {
26        "obj_name": "is_directive_arg",
27        "arg_index": 0,
28        "is_entry_types": ["DirectiveString"],
29        "is_arg_type": "String"
30      }
31    ],
32  "description": "The TTL directive inserts a title at the start of
each page of a listing file."
33 },
34 ...

```

An example, abridged, listing of `is_directive` JSON object entries.

An `is_directive` object entry is somewhat similar to the opcode JSON object's we've seen before. They can be thought of as a simplified version of the `is_op_code` object. Direct your attention to the first `is_directive` listed, the `@ORG` directive. We'll quickly review the object's attributes here. The first attribute is the `obj_name` attribute which holds a string representing the proper name of the object. The next attribute is the `directive_name` attribute which holds the actual name of the directive as it appears in assembly source code.

Subsequently, the `arg_len` attribute indicates the length of the `args` attribute's array. Similar to what we've seen with opcodes the directives have a list of arguments associated with them. The `is_directive_arg` JSON object is very similar to the `is_op_code_arg` objects we've already reviewed but we'll still spend some time reviewing them here.

*Listing 7-24: Example is\_directive\_arg Object 1*

```
01 {
02     "obj_name": "is_directive_arg",
03     "arg_index": 0,
04     "bit_index": 0,
05     "is_entry_types": ["Number"],
06     "is_arg_type": "Number",
07     "bit_series": {"bit_start": 0, "bit_stop": 31, "bit_len": 32},
08     "num_range": {"min_value": 0, "max_value": 2147483647, "bit_len": 32,
09     "twos_complement": false, "alignment": "WORD"},
10     "bit_shift": {"shift_dir": "NONE", "shift_amount": -1}
11 }
```

An example `is_directive_arg` object from the `is_directives.json` file.

Notice the similarity between the `is_directive_arg` object and the `is_op_code_arg` objects we've seen previously. The `obj_name` attribute holds a string representing the proper object name. The `arg_index` attribute indicates which position in the assembly source code syntax the argument appears. The `bit_index` attribute is similar to the `arg_index` attribute except it represents the position in the binary encoding where the argument appears.

The next attribute to discuss is the `is_entry_types` attribute. This attribute is very similar to attributes we've seen with a similar name. The attribute holds an array of `is_entry_type` names that can be used to satisfy the directive argument requirement. The next attribute listed is the `is_arg_type` attribute. Notice that it has an entry type name as a value. Can you think of a reason why this is?

The main reason is that the opcodes argument data types are defined by the instruction set definition and are converted to the assembler's

supported data types. The same situation does not exist for standard directives because they are defined as part of the assembler itself. For that reason, the `is_arg_type` attribute will not have similar values as the similarly named attribute found on opcode objects.

*Listing 7-25: Example is\_directive\_arg Object 2*

```
1 {
2   "obj_name": "is_directive_arg",
3   "arg_index": 0,
4   "is_entry_types": ["DirectiveString"],
5   "is_arg_type": "String"
6 }
```

An example `is_directive_arg` object from the `is_directives.json` file.

A second `is_directive_arg` example is provided in the previous listing. The attributes listed are part of the objects we've seen before, so I won't review them again here. Take a look at the second example provided and make sure you understand it. Also, take a moment to review the `is_directives.json` file. We've reviewed directives, in detail, in Chapter 6, but please review this data file on your own and make sure you understand it before moving on.

## **JSON Data Files: Example Source Lines: `is_empty_data_lines.json`**

The last JSON data file for us to look at is the `is_empty_data_lines.json` file. This file is very complex, so we won't be listing it here in full. We'll use a heavily abridged version. The file contains a JSON object that is a copy of a valid empty data directive line of assembly code. The reason why this information is being held in a JSON data file is that it simplifies the implementation. We can create a new line of assembly code, even after parsing and mapping the assembly source code with JSON objects.

When we need to adjust the alignment of a data area during the assembly process, we can inject this JSON data to get it done. Another

solution to this problem would have been to add clone support to all the classes involved so that we could clone ourselves a new data line. That would add a lot of new code and complexity, so we decided to just data drive the lines we need to be able to create, and as such we have the `is_empty_data_lines.json` data file. Let's take a look at the main JSON object sans the data line details.

*Listing 7-26: JSON Data File: is\_empty\_data\_lines.json*

```
01 {  
02     "obj_name": "is_empty_data_lines",  
03     "set_name": "THUMB_ARM7TDMI",  
04     "is_empty_data_lines": [  
05         {  
06             "obj_name": "TokenLine",  
07             "payloadDirective": "@DCHW",  
08             "payloadBinRepStrEndianBig1": "0000000000000000",  
09             "payloadBinRepStrEndianLill1": "0000000000000000",  
10             ...  
11         }  
12     ]  
13 }
```

An example of the `is_empty_data_lines` JSON object with many details removed.

We'll review the attributes that are visible in the example listed previously. The `obj_name` and `set_name` attributes are familiar to us, so we'll move past them. Next, the `is_empty_data_lines` attribute is an array which holds the JSON data representation of a parsed assembly code line that's an empty data directive line. Take a look at the object definition on lines 5 and 6. Notice that the `obj_name` attribute for this object is "TokenLine".

The pattern of the `obj_name` attribute is indicative of a programmatically generated JSON objects, like the kind that are created when the assembler parses source code. That's the most we can discuss about this data file at this time but it's use as a shortcut will become more apparent when we see it used by the code. That brings us to the conclusion of this JSON data file review.

# Chapter Conclusion

In this chapter we completed a detailed review of the JSON data files that provide all of the data necessary to describe the ARM Thumb-1 instruction set and all of the assembler's supported standard directives with regard to the Thumb-1 instruction set. These data files are extremely important and it's crucial that you get a good understanding of the structure of each one.

Recall, these files are all associated with the instruction set we're working on even though they define assembler level functionality like standard directives. Let's list the JSON data files that we've reviewed in this chapter and briefly describe the data they contain.

- **Instruction Sets:** `is_sets.json`: The main JSON data file that describes all the files associated with a given instruction set. The file definitions also include information about which Java class, fully qualified package name, is used to load the specified data file. This means the file loading and Java class utilization is totally data driven.
- **Argument Types:** `is_arg_types.json`: A JSON data file that describes the different argument data types defined by the given instruction set, in our case ARM Thumb-1. The objects in this data file provide a conversion between the instruction set's defined argument data types and the assembler's concept of argument data types which are closer to Java data types for obvious reasons.
- **Entry Types:** `is_entry_types.json`: A very important JSON data file that describes the different types of entries that can exist in the assembly source code file. An entry type is a known piece of text, or in other words a token. Recall that the assembly source code file is lexized into artifacts, unidentified strings, which are then tokenized into tokens, known pieces of text with attributes. Then `is_entry_types.json` data file handles converting the artifacts to tokens.
- **OpCodes:** `is_opcodes.json`: The `is_opcodes` JSON data file is an integral part of the Thumb-1 instruction set implementation as it describes in detail all the supported opcodes and their arguments. When the assembler is processing a Thumb-1 assembly source code file it parses the text ultimately resulting in a JSON representation of the

source code which is compared to data files like `is_opcodes.json`.

- **Registers:** `is_registers.json`: A simple data file that describes in detail all the registers that are available for the Thumb-1 instruction set including any aliases for high registers, etc.
- **Valid Source Lines:** `is_valid_lines.json`: A JSON data file that provides structure to our instruction set implementation by describing what assembly source code lines are valid for the Thumb-1 instruction set.
- **Directives:** `is_directives.json`: A data file that describes in detail the standard directives and their arguments that are supported by the assembler when processing ARM Thumb-1 assembly source code files.
- **Example Source Lines:** `is_empty_data_lines.json`: A data file that provides a template for an assembly data directive line. This data is used to inject innocuous lines of assembly source code to offset data areas, so they have the proper byte alignment.

We should take a step back and talk about the bigger picture. The data files we've defined here are our implementation of the Thumb-1 instruction set, opcodes, registers, directives, line syntax, and grammar. If you implemented the same instruction set, you would have maybe a similar but different set of data files and objects. That is to be expected. What is important to realize here is that the assembler does not put any constraints on the structure of the data files, object, or Java classes you decide to use to define your instruction set. We'll explore the concepts of extensibility further in the next chapter.

# **Chapter 8: Simple, Extensible Coding Model**

We've covered a lot of material up to this point in the text. For the most part that material has been geared to defining the instruction set. Chapter 8 marks the start of our review of the GenAsm assembler's code. We're going to talk about the directory structure of the project and data files we've just reviewed followed by an introduction to the code that powers the assembler.

Lastly, we'll talk about the extensibility of the code model and how we can create custom JSON data files and Java classes to support them and bring new information into an instruction set implementation. I should mention that at times I'll refer to the assembler when I really mean the Thumb-1 instruction set implemented in the GenAsm assembler. Almost everything we do with regard to the assembler and its code base requires an instruction set to work on so the lines between the two concepts tend to blur for me.

Take the time to open up and quickly review the files as they are introduced and discussed in the following sections. This will give you some added exposure to the project's code, even if it is just in passing. That's fine, we'll be reviewing the assembler's classes, in detail, in the upcoming chapters.

# Structure of Data Files and Folders

In this section we're going to discuss the structure of the data files and folders used to drive a specific instruction set implementation. Always keep in mind that we're working on a specific implementation but the assembler is designed to be extensible so you can add new instruction sets to it by defining new JSON data files and Java classes to handle them.

First off, recall that the GenAsm NetBeans project has its working directory set to the NetBeans' project folder. This means that the "cfg" directory should be in the local folder and accessible with the local folder path, ".\cfg".

*Listing 8-1: Directory and File Structure of the ARM Thumb-1 Implementation*

```
GenAsm          (NetBeans Project Root)
-> cfg          (Assembler config folder)
  -> is_sets.json
  -> THUMB        (ARM Thumb-1 Implementation)
    -> is_arg_types.json
    -> is_directives.json
    -> is_empty_data_lines.json
    -> is_entry_types.json
    -> is_op_codes.json
    -> is_registers.json
    -> is_valid_lines.json
    -> is_test_file_vals.json
    -> EMUS          (Emulators)
    -> INCBIN        (File Includes)
    -> OUTPUT         (Executed Program Output)
    -> TEST          (Test Programs)
```

A listing of the project's cfg folder with ARM Thumb-1 implementation.

The folder structure of the NetBeans project demonstrates the default Thumb-1 implementation. One key thing to note is that the is\_sets.json data file exists outside of any instruction set implementation and it has a static implementation meaning that it's not expected to change or be customized.

That's not to say that you can't do so, it's just designed to allow for the addition of new instruction sets without changing the `is_sets.json` implementation. This gives us a foundation to build off of.

You should also note that the Thumb-1 implementation defines the data files that it relies on in the `is_sets.json` data file. This includes the path to the JSON data file. You can see that we've structured things so that the data files reside in the root of the instruction set implementation's directory.

In order to add a new instruction set, to the GenAsm assembler you would create a new folder in the config root directory to keep your data files, and test program folders, etc. Notice that in our implementation we have special folders like the `EMUS` directory that holds our test emulators. The `INCBIN` folder holds all of the "include" files that are used by the test programs.

The `OUTPUT` folder is used to hold and organize all of the output files generated by the assembler when a test program is executed. Lastly, the `TEST` folder is used to hold and organize all the test programs that are used to verify the assembler's functionality with regard to the current instruction set. In the next section we'll take a look at the structure of the Java classes that power the assembler.

## Structure of the Project's Java Classes

The GenAsm assembler's Java classes are structured to allow for extension of the code to support new instruction sets and to make common functionality available to new instruction sets. First let's take a look at the base functionality provided by the code base. The base functionality is considered common code and isn't necessarily associated with any instruction sets. I'll list the Java namespaces here and provide a general description of their use.

- **net.middlemind.GenAsm:** The main entry point into the GenAsm project this package includes the static main class and some very general, high-level, classes.

- **net.middlemind.GenAsm.Assemblers:** A package that provides generic support for assemblers.
- **net.middlemind.GenAsm.Exceptions:** A package that provides generic support for exceptions used throughout the project.
- **net.middlemind.GenAsm.FileIO:** A generic package that provides file IO support for the project.
- **net.middlemind.GenAsm.JsonObjs:** A package that provides support for JSON object classes. JSON object classes are used to hold the data that is loaded from the JSON data files.
- **net.middlemind.GenAsm.Lexers:** A package that provides general support for lexerizing assembly source code files.
- **net.middlemind.GenAsm.Linkers:** A package that provides general support for linkers.
- **net.middlemind.GenAsm.Loaders:** The base package for all JSON data file loaders. This package is used to load JSON data files into a target JsonObj holder classes.
- **net.middlemind.GenAsm.PreProcessors:** A package that provides general preprocessor support for the assembler. Since the preprocessor is defined in code its support could be potentially applied to any new instruction sets, however, this may require some more customization of the class.
- **net.middlemind.GenAsm.Tokeners:** A package that provides general support for tokenizing a lexerized assembly source code file.

There's a lot going on here so let's review each package in detail and take a quick look at the associated Java classes. Read through the subsequent list of files and make sure you have an idea of what each Java class does. Please take the time to quickly checkout each file as you go through the list before moving on.

- **GenAsm:**
  - **GenAsm.java:** Static main entry point of the GenAsm assembler.
  - **Logger.java:** A static utility class that provides logging support.
  - **Utils.java:** A static utility class that provides varied general support.
- **Assemblers:**
  - **Assembler.java (Interface):** An interface for an assembler implementation. Defines the minimum set of methods to implement to plug in a new assembler implementation.
  - **AssemblerEventHandler.java (Interface):** An interface that defines callback methods that can be used to customize an existing assembler implementation.
  - **Symbol.java:** A class that represents a symbol like those created by using labels and the `@EQU` directive in assembly source code.
  - **Symbols.java:** A class that is used to manage the symbols associated when parsing an assembly program.
  - **TwosComplement.java:** A class that provides assistance with applying the two's complement to a binary number.
- **Exceptions:**
  - **ExceptionBase.java:** A class that's used as the base class of all custom exceptions used in the project.
- **FileIO:**
  - **FileLoader.java:** A static class that provides general support for loading files of different types into classes.
  - **FileUnloader.java:** A static class that provides general support for unloading classes to files.
- **JsonObjs:**

- **JsonObj.java (Interface):** The interface used to define the basic requirements necessary to implement a JsonObj, holder class.
  - **JsonObjBase.java:** The base class of all JSON object classes.
  - **JsonObjBitRep.java:** A JSON object holder class that represents a bit representation sub-object.
  - **JsonObjBitSeries.java:** A JSON object holder class that represents a bit series sub-object.
  - **JsonObjBitShift.java:** A JSON object holder class that represents a bit shift sub-object.
  - **JsonObjIlsSet.java:** A JSON object holder class that represents an instruction set object.
  - **JsonObjIlsSets.java:** A JSON object holder class that is used to hold a set of `JsonObjIlsSet` object instances.
  - **JsonObjLineHexRep.java:** A JSON object holder class that represents a line hex representation set object.
  - **JsonObjLineHexReps.java:** A JSON object holder class that holds a set of `JsonObjLineHexRep` object instances.
  - **JsonObjNumRange.java:** A JSON object holder class that represents a valid number range sub-object.
  - **JsonObjTxtMatch.java:** A JSON object holder class that represents a valid text match sub-object.
- **Lexers:**
    - **Artifact.java:** A class used to model an artifact, a contiguous string of text, generated by the lexer.
    - **ArtifactLine.java:** A class used to model the artifacts found on a line of assembly source code.

- **Lexer.java (Interface):** An interface that defines the minimum set of methods needed to implement a lexer that plugs into the assembler.
- **Linkers:**
  - **Linker.java (Interface):** An interface that defines the minimum set of methods needed to implement a linker that plugs into the assembler.
- **Loaders:**
  - **Loader.java (Interface):** An interface that defines the minimum set of methods needed to implement a JSON loader.
  - **LoaderBitSeries.java:** A JSON loader that is designed to load bit series sub-objects.
  - **LoaderIsSets.java:** A JSON loader that is designed to load an is\_set.json data file.
  - **LoaderLineHexReps.java:** A JSON loader that is designed to load a hex representation file. These files are found as part of some of the test programs and are used in the assembler's unit tests.
- **PreProcessors:**
  - **PreProcessor.java (Interface):** An interface class that defines how to implement a class that handles preprocessor directives.
- **Tokeners:**
  - **Token.java:** A class that represents a token. A token is a recognized artifact, or an identified text segment. A text segment can be recognized as any of the types defined in the is\_entry\_types.json data file.
  - **TokenLine.java:** A class that represents the tokens extracted from a line of assembly source code.

- **TokenSorter.java**: A class used to sort tokens by index ascending or descending.
- **Tokener.java (Interface)**: An interface that defines the minimum set of methods needed to implement a tokenizer that will plug into the assembler.

Next, we'll continue our exploration of the project's source code by listing all of the classes that are used to implement that ARM Thumb-1 instruction set in the GenAsm assembler. I'll omit the majority of the exception classes due to their redundancy. A few other class types will be abridged. Please take the time to look over the classes listed here and the ones that are not. We'll review them, in detail, in the coming chapters but it's good to get as much exposure to them as possible.

- **Assembler.Thumb**:
  - **AreaThumb.java**: A class that describes an area of assembly code as defined by standard directives.
  - **AssemblerEventHandlerThumb.java**: A class that implements callback methods implemented before and after key method calls.
  - **AssemblerThumb.java**: An assembler implementation for the ARM Thumb-1 instruction set.
  - **BuildOpCodeEntryThumbSorter.java**: A class used to sort a list of `BuildOpCodeThumb` classes.
  - **BuildOpCodeThumb.java**: A class that represents a built, binary encoded, thumb opcode.
- **Exceptions.Thumb**:
  - **ExceptionDirectiveArgNotSupported.java**: An exception class that fires when a directive argument is not supported.

- **ExceptionInvalidArea.java:** An exception that fires when an invalid area definition is encountered in the source code.
  - ...
- **JsonObjs.Thumb:**
  - **JsonObjIsArgType.java:** A class designed to hold opcode argument type object data.
  - **JsonObjIsArgTypes.java:** A class designed to hold an array of `JsonObjIsArgType` object instances.
  - ...
- **Lexers.Thumb:**
  - **LexerThumb.java:** An implementation of the `Lexer.java` interface designed to work with the ARM Thumb-1 instruction set.
- **Linkers.Thumb:**
  - **LinkerThumb.java:** An implementation of the `Linker.java` interface designed to work with the objects resulting from the Thumb-1 assembler.
- **Loaders.Thumb:**
  - **LoaderIsArgTypes.java:** A loader class designed to parse and load opcode argument type JSON data file.
  - **LoaderIsDirectives.java:** A loader class designed to parse and load directive JSON data file.
  - **LoaderIsEmptyDataLines.java:** A loader class designed to parse and load the empty data lines JSON data file.
  - ...
- **PreProcessors.Thumb:**
  - **PreProcessorThumb.java:** An implementation of the `PreProcessor.java` interface designed to handle preprocessor directives for the ARM Thumb-1 instruction set.
- **Tokeners.Thumb:**

- **TokenerThumb.java:** An implementation of the Tokener.java interface designed to work with the Thumb-1 instruction set.

Make sure to take a quick look at each file as you read through them. Notice that there is an implementation of each key class for the Thumb-1 instruction set. Each of these key classes extends a super class in the preceding list of base classes. Also, take a moment to look at the sections of classes that have been abridged and make sure you are familiar with them. Think about what loaders, JSON objects, and exceptions there are and try to imagine what they're used for.

## Adding a Custom Data File to an Instruction Set

In this section we'll discuss how to add a new JSON data file to an instruction set including both a JSON object holder and a JSON object loader. The classes we'll look at in this section are not detailed reviews. They are just direct listings. We'll introduce a detailed class review methodology when we get into the code review chapters. Let's take a moment to define the contents of the new data file we want to load. We'll create a very simple one that has a few attributes. Let's define the JSON data file here.

*Listing 8-2: Definition of the new JSON Data File*

```

01 {
02   "obj_name": "is_test_file_vals",
03   "set_name": "THUMB_ARM7TDMI",
04   "is_test_file": true,
05   "is_test_file_vals": [
06     {
07       "obj_name": "is_test_file_val",
08       "is_fake_data": true
09     }
10   ]
11 }
```

An example of a new JSON data file to add to the Thumb-1 instruction set.

The previously shown listing is an example of a test JSON data file that we can “add” to the Thumb-1 instruction set implementation. The file can be found at the following location off of the root of the project’s directory structure. I’ll list the relative path here.

```
.\cfg\THUMB\is_test_file_vals.json
```

The JSON data file is exceedingly simple but the process for adding it to the instruction set implementation’s list of data files we’ll need to have a few Java classes created and ready to go. The first such class is one that can hold the parsed data from the test JSON data file. We’ll call this class `JsonObjIsTestFileVals` and we’ll use it to hold the data loaded from the main data file. We’ll also need a second class to hold the data from parsing `is_test_file_val` child objects.

We’ll call this class, `JsonObjIsTestFileVal`. Note that this follows a similar pattern to the parent-child JSON data loading that other instruction set data files use. On top of the files needed to hold the data we intend to load we also need a JSON data file loader. The loader we’ll use for this example is named the `LoaderIsTestFileVals` class.

Now that we have the JSON data file defined, and the loader, holder classes thought out let’s flush out the Java classes we need to add this JSON data file to the instruction set’s list of files. In the next section we’ll take a look at the Java classes that will hold the parsed JSON data.

## Creating a Custom JSON Object Holder

In this section we will quickly, and in no great detail, review a simple class designed to hold the data from the main object in the test JSON data file. Notice that in the subsequent listing the `JsonObjIsTestFileVals` class extends the super class, `JsonObjBase`. Take a moment to look over the following listing.

*Listing 8-3: Definition of the `is_test_file_vals.json` Data File’s Main Object Holding Class*

```
01 public class JsonObjIsTestFileVals extends JsonObjBase {  
02     public String obj_name;
```

```

03     public String set_name;
04     public boolean is_test_file;
05     public List<JsonObjIsTestFileVal> is_test_file_vals;
06
07     @Override
08     public void Print() {
09         Print("");
10    }
11
12     @Override
13     public void Print(String prefix) {
14         super.Print(prefix);
15         Logger.wrl(prefix + "ObjectName: " + obj_name);
16         Logger.wrl(prefix + "SetName: " + set_name);
17
18         Logger.wrl(prefix + "IsTestFileVals:");
19         for(JsonObjIsTestFileVal entry : is_test_file_vals) {
20             Logger.wrl("");
21             entry.Print(prefix + "\t");
22         }
23     }
24 }
```

A listing of the JsonObjIsTestFileVals class designed to hold the data contained by the is\_test\_file\_vals.json data file.

Take a moment to look over the class' fields. Note that there is at least a field for each attribute present on the main object of the is\_test\_file\_vals.json data file. That's as simple as it is. One caveat is that the array attribute, is\_test\_file\_vals, requires a List of a certain object type. Namely, the object designed to hold the is\_test\_file\_val JSON object data. In this case we have another Java class designed to hold the data contained in this JSON sub-object, the JsonObjIsTestFileVal class.

***Listing 8-4: Definition of the is\_test\_file\_vals.json File's Child Object Holding Class***

```

01 public class JsonObjIsTestFileVal extends JsonObjBase {
02     public String obj_name;
03     public boolean is_fake_data;
04 }
```

```

05     @Override
06     public void Print() {
07         Print("");
08     }
09
10    @Override
11    public void Print(String prefix) {
12        super.Print(prefix);
13        Logger.wrl(prefix + "ObjectName: " + obj_name);
14        Logger.wrl(prefix + "IsFakeData: " + is_fake_data);
15    }
16 }
```

A listing of the `JsonObjIsTestFileVal` class designed to hold the data contained by the main JSON object's `is_test_file_vals` array attribute.

The `JsonObjIsTestFileVal` is very similar to the parent class, `JsonObjIsTestFileVals`, that we've just reviewed. Take a moment to note the naming convention used throughout the JSON object holder classes. The main JSON object is usually in the Java class with the pluralized name, while the sub-object is in the class with the singular name. I use this pattern a lot in this project.

You should also notice that the attributes of the JSON sub-object, `is_test_file_val`, are represented as class fields. That takes care of a quick review of the Java data holder classes. They are fairly straight forward and don't require a lot of coding to setup. In the next section we'll take a look at a JSON data loading class, colloquially referred to as a loader.

## Creating a Custom JSON Object Loader

The class responsible for loading the JSON data file, `is_test_file_vals.json`, is the `LoaderIsTestFileVals` class. This class is JSON data loader class, and it follows the same naming conventions and structure as the other JSON data loading classes in the project. Let's take a look at some code!

*Listing 8-5: Definition of the `is_test_file_vals.json` Data File's Loading Class*

```
01 public class LoaderIsTestFileVals implements Loader {
```

```

02     public String obj_name = "LoaderIsTestFileVals";
03
04     @Override
05     public JsonObjIsTestFileVals ParseJson(String json, String
targetClass, String fileName) throws ExceptionLoader {
06         GsonBuilder builder = new GsonBuilder();
07         builder.setPrettyPrinting();
08
09         Gson gson = builder.create();
10         try {
11             JsonObjIsTestFileVals jsonObj =
(JsonObjIsTestFileVals)Class.forName(targetClass).getConstructor().newInstance();
12             jsonObj = gson.fromJson(json, jsonObj.getClass());
13             jsonObj.name = targetClass;
14             jsonObj.fileName = fileName;
15             jsonObj.loader = getClass().getName();
16
17             for(JsonObjIsTestFileVal entry : jsonObj.is_test_file_vals) {
18                 entry.name = entry.getClass().getName();
19                 entry.fileName = fileName;
20                 entry.loader = getClass().getName();
21             }
22
23             return jsonObj;
24         } catch (ClassNotFoundException | NoSuchMethodException |
InstantiationException | IllegalAccessException | InvocationTargetException e) {
25             throw new ExceptionLoader("Could not find target class, " +
targetClass + ", in loader " + getClass().getName());
26         }
27     }
28 }
```

A full listing of the LoaderIsTestFileVals JSON data loader class.

Make sure to look over the class listed previously. Don't worry if the class seems confusing to you, it's a bit complex and is deserving of a detailed review. For our purposes here we're really only interested in seeing how this class utilizes the classes defined in the previous section. If you direct your

attention to lines 11 - 12 you'll see the main JSON data class is populated from a call to the `fromJson` method.

Also notice that on line 17 the `is_test_file_vals` class field is already populated as the JSON objects were parsed and loaded as part of the main JSON object. That brings us to the conclusion of this section. In the next section we'll briefly take a look at how the GenAsm project's static main class is called from the command line.

## Registering the New Data File with an Instruction Set

To test the adding the new JSON data file to the Thumb-1 instruction set look at the following example file.

```
.\cfg\storage\is_sets_demo.json
```

Now copy the last entry from the list of objects. I'll list it here for convenience.

*Listing 8-6: New is\_file Entry Pointing to the New JSON Data File*

```
{
  "obj_name": "is_file",
  "path": "./cfg/THUMB/is_test_file_vals.json",
  "loader_class":
"net.middlemind.GenAsm.Loaders.Thumb.LoaderIsTestFileVals",
  "target_class":
"net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsTestFileVals",
  "category": "special"
}
```

A new `is_file` entry to demonstrate adding a new JSON file to the instructions set's list of files.

Copy and paste the `is_file` object and add it to the list in the `is_sets.json` file. Be sure to include a comma in the file so that the array's syntax is correct. Now if you run the `GenAsm.java` with no arguments the new

file will be loaded automatically as part of the instruction set. The subsequent listing shows the program's output indicating that the new data file has been loaded.

*Listing 8-7: IDE Output Showing Newly Added JSON Data File Loaded*

```
AssemblerThumb: RunAssembler: Json loaded
'./cfg/THUMB/is_test_file_vals.json'
AssemblerThumb: RunAssembler:
net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsTestFileVals
AssemblerThumb: RunAssembler: Json parsed as
'net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsTestFileVals'
AssemblerThumb: RunAssembler: Loading isaData with entry
'net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsTestFileVals'
```

An example of GenAsm.java assembling a Thumb-1 program and indicating the new JSON data file loaded correctly.

Make sure to return the `is_files.json` file to its original state before moving onto the next section. You can reference the following backup file if you are unsure what to do to restore it. You can also just download a fresh copy of it from the code repository.

`.\cfg\storage\is_sets_orig.json`

That wraps up our little demonstration section showing how easy it is to add new JSON data files to the assembler, specifically to an instruction set's file list. In the next section we'll take a look at how you can call the program via the command line and provide arguments that define how much of the assembler functions.

## Calling the Static Main

In this section, as you might have guessed, we're going to be looking at how the GenAsm project's static main class can be called to execute a target assembly source code file and generate a binary executable, among other files. There are two ways to call the project's

static main class. One way is to provide a set of command line arguments to describe what program to run, what instruction set to use, and a few other settings.

The second way to call the program is to pass no command line arguments to the program which will cause the use of default values that have been hard coded in the GenAsm class. As I've mentioned before we'll cover this class in much more detail a little later on in the text. For now, I just want to focus on the different ways to execute the assembler and provide arguments to it. Let's take a look at the first execution method now.

*Listing 8-8: Assembler Execution – Command Line Arguments Example*

```
//Execution Template
java -jar GenAsm.jar [sets file name] [target set] [sets loader class]
[sets holder class] [assembler class] [assembler source code file] [linker
class] [preprocessor class] [root output directory] [verbose] [quell file
output] [config dir path]

//Execution Specific
java -jar GenAsm.jar
00  "..\cfg\is_sets.json"
01  "THUMB_ARM7TDMI"
02  "net.middlemind.GenAsm.Loaders.LoaderIsSets"
03  "net.middlemind.GenAsm.JsonObjs.JsonObjIsSets"
04  "net.middlemind.GenAsm.Assemblers.ThumbAssemblerThumb"
05  "..\cfg\THUMB\TESTS\TEST_N_AsmChecks\genasm_source.txt"
06  "net.middlemind.GenAsm.Linkers.ThumbLinkerThumb"
07  "net.middlemind.GenAsm.PreProcessors.ThumbPreProcessorThumb"
08  "..\cfg\THUMB\OUTPUT\TEST_N_AsmChecks\";
09  false
10  false
11  "..\cfg\"
```

An example of the GenAsm command line execution with provided arguments shown as a template and as an actual example. Please keep in mind the working directory when testing assembler execution manually via a terminal or console. The index of the argument has been added on new lines for clarity.

Another example of calling the GenAsm assembler, but from another program instead of via the command line is shown here.

```
public static String CFG_DIR = "C:\FILES\DOCUMENTS\GitHub\GenAsm\cfg\";  
public String[] mainExeArgs = new String[GenAsm.ARG_LEN_TARGET];  
public String testName = "TEST_L_HelloWorld";  
public String cfgDir = CFG_DIR;  
public String assemblySourceFile = cfgDir + "THUMB\\TESTS\\" + testName +  
"\genasm_source.txt";  
  
mainExeArgs[0] = cfgDir + "is_sets.json";  
mainExeArgs[1] = "THUMB_ARM7TDMI";  
mainExeArgs[2] = "net.middlemind.GenAsm.Loaders.LoaderIsSets";  
mainExeArgs[3] = "net.middlemind.GenAsm.JsonObjs.JsonObjIsSets";  
mainExeArgs[4] = "net.middlemind.GenAsm.Assemblers.ThumbAssemblerThumb";  
mainExeArgs[5] = assemblySourceFile;  
mainExeArgs[6] = "net.middlemind.GenAsm.Linkers.ThumbLinkerThumb";  
mainExeArgs[7] =  
"net.middlemind.GenAsm.PreProcessors.ThumbPreProcessorThumb";  
mainExeArgs[8] = cfgDir + "THUMB\\OUTPUT\\" + testName + "\\\";  
mainExeArgs[9] = "false";  
mainExeArgs[10] = "false";  
mainExeArgs[11] = cfgDir;  
  
GenAsm.main(mainExeArgs);
```

As you can see from the previously listed examples there are quite a few required arguments to keep track of if you plan on calling the program either way. Now that is why we added a hard-coded, no argument, option. It really comes in handy during the development of a new instruction set implementation or a complex assembly program. Take a minute to look over the different argument types and the example values provided in the previous listing. Does anything stand out to you?

Along with the ability to define new JSON data files including their associated Java classes as part of the instruction set's file definitions. You

can specify the classes used to power the following aspects of the assembler. The only requirement is that they extend and properly implement the correct interfaces. This should give developers great power and flexibility in how they implement their instruction set by allowing them to customize almost all aspects of the assembler. Let's take a look at some of the code that controls the assembler's argument handling.

*Listing 8-9: Assembler Execution – Command Line No-Arguments Example*

```
01 } else if(args == null || args.length < ARG_LEN_TARGET) {  
02     String targetProgram = "TEST_A_AllOpCodes";  
03     CFG_DIR_PATH = ".\\cfg\\\";  
04     ASM_SETS_FILE_NAME = "./cfg/is_sets.json";  
05     ASM_TARGET_SET = "THUMB_ARM7TDMI";  
06     ASM_SETS_LOADER_CLASS =  
"net.middlemind.GenAsm.Loaders.LoaderIsSets";  
07     ASM_SETS_TARGET_CLASS =  
"net.middlemind.GenAsm.JsonObjs.JsonObjIsSets";  
08     ASM_SETS = null;  
09     ASM_SET = null;  
10     ASM_ASSEMBLER_CLASS =  
"net.middlemind.GenAsm.Assemblers.ThumbAssemblerThumb";  
11     ASM_ASSEMBLER = null;  
12     ASM_ASSEMBLY_SOURCE_FILE = CFG_DIR_PATH + "THUMB\\TESTS\\" +  
targetProgram + "\\genasm_source.txt";  
13     ASM_LINKER_CLASS =  
"net.middlemind.GenAsm.Linkers.ThumbLinkerThumb";  
14     ASM_LINKER = null;  
15     ASM_PREPROCESSOR_CLASS =  
"net.middlemind.GenAsm.PreProcessors.ThumbPreProcessorThumb";  
16     ASM_PREPROCESSOR = null;  
17     ASM_ROOT_OUTPUT_DIR = CFG_DIR_PATH + "THUMB\\OUTPUT\\" +  
targetProgram + "\\\";  
18     ASM_VERBOSE = false;  
19     ASM_QUELL_FILE_OUTPUT = false;  
20 } else if(args != null || args.length == ARG_LEN_TARGET) {  
21     CFG_DIR_PATH = args[11];  
22     ASM_SETS_FILE_NAME = args[0];  
23     ASM_TARGET_SET = args[1];  
24     ASM_SETS_LOADER_CLASS = args[2];  
25     ASM_SETS_TARGET_CLASS = args[3];  
26     ASM_SETS = null;
```

```
27     ASM_SET = null;
28     ASM_ASSEMBLER_CLASS = args[4];
29     ASM_ASSEMBLER = null;
30     ASM_ASSEMBLY_SOURCE_FILE = args[5];
31     ASM_LINKER_CLASS = args[6];
32     ASM_LINKER = null;
33     ASM_PREPROCESSOR_CLASS = args[7];
34     ASM_PREPROCESSOR = null;
35     ASM_ROOT_OUTPUT_DIR = args[8];
36     ASM_VERBOSE = Boolean.parseBoolean(args[9]);
37     ASM_QUELL_FILE_OUTPUT = Boolean.parseBoolean(args[10]);
38 }
```

A code listing showing the GenAsm class' static main entry point code for handling the program's command line arguments.

With regard to the argument-based execution of the GenAsm program direct your attention to the segment of code on lines 20 – 37. Each argument provided is mapped to a static class field and used to drive the assembler. Similar functionality can be attained with hard coded values, lines 1 – 19, if you're working on a particular program and want to run it without messing around with the command line arguments.

Notice that the way the test programs are setup it's easy to change to a different program. All you have to do is change the name of the variable, targetProgram, on line 2. I recommend keeping to the structure that the test programs use. It'll save you time and it's quicker and easier to use this way. In the next section we'll take a close look at the output files generated by the GenAsm assembler.

## Breakdown of the Output Files

Because this assembler, including Thumb-1 instruction set implementation, is designed for learning purposes it generates a lot of output files. These can be used to track the changes the assembler is making by validating them in the resulting output files.

- **output\_area\_desc\_code.json**: A description of the code area, if any, in the assembly program.
- **output\_area\_desc\_data.json**: A description of the data area, if any, in the assembly program.
- **output\_area\_lines\_code.json**: A collection of the lines of assembly code if any, that are in the code area.
- **output\_area\_lines\_data.json**: A collection of the lines of assembly code, if any, that are in the data area.
- **output\_assembly\_listing\_endian\_big.bin**: A binary representation of the final program in big endian encoding.
- **output\_assembly\_listing\_endian\_big.list**: A listing file of the assembly program in big endian encoding.
- **output\_assembly\_listing\_endian\_lil.bin**: A binary representation of the final program in little endian encoding.
- **output\_assembly\_listing\_endian\_lil.list**: A listing file of the assembly program in big endian encoding.
- **output\_lexer.json**: A representation of the assembly program with adjustments made by the lexer.
- **output\_linked\_area\_lines\_code.json**: A JSON data file showing the lines associated with a given area definition, code or data.
- **output\_pre\_processed\_assembly.txt**: A text file that's an update of the assembly program with adjustments made by the preprocessor.
- **output\_symbols.json**: A summary of the symbols found in the assembly program. Symbol entries are created by labels and @EQU directives.

- **output\_tokened\_phase0\_tokenized.json**: A representation of the program after the lexer runs and the output\_lexer.json files is generated.
- **output\_tokened\_phase1\_valid\_lines.json**: A representation of the program after the validate tokenized line check is run.
- **output\_tokened\_phase2\_refactored.json**: A representation of the program after comments, lists, and groups are collapsed, register ranges are expanded, opcodes and directives are populated.
- **output\_tokened\_phase3\_valid\_lines.json**: A representation of the program after opcodes and directives have been validated and the output\_symbols.json file has been written.
- **output\_tokened\_phase4\_bin\_output.json**: A representation of the program after all steps have been processed. This will include final binary representations of all valid lines.

I want to also list the steps, in pseudo code, that are taken when the assembler is running so you can see what's going on in-between when each output file is written.

- **LexerizeAssemblySource**  
Write: output\_lexer.json
- **TokenizeLexerArtifacts**  
Write: output\_tokened\_phase0\_tokenized.json
- **ValidateTokenizedLines**  
Write: output\_tokened\_phase1\_valid\_lines.json
- CollapseCommentTokens
- ExpandRegisterRangeTokens

- CollapseListAndGroupTokens
- PopulateOpCodeAndArgData
- **PopulateDirectiveArgAndAreaData**  
Write: output\_tokened\_phase2\_refactored.json
- ValidateOpCodeLines
- **ValidateDirectiveLines**  
Write: output\_tokened\_phase3\_valid\_lines.json  
Write: output\_symbols.json
- **BuildBinLines**  
Write: output\_tokened\_phase4\_bin\_output.json

Make sure to look over the previous listing and familiarize yourself with the general flow of steps. It's okay if some of the method calls used in a particular step aren't clear, we'll review the code in detail soon.

## Chapter Conclusion

In this chapter we've completed a high-level, general, review of the project's extensibility model. This aspect of the project allows the developer to extend the program by adding new files to support functionality in one or more supported instruction sets.

- **Structure of Data Files and Folders:** A review of the project's directory structure including key data files for the Thumb-1 instruction set.
- **Structure of the Project's Java Classes:** A review of the generic base classes, interfaces and other associated Java classes, and the classes that support the Thumb-1 instruction set.
- **Adding a Custom Data File to an Instruction Set:** An example demonstrating adding new JSON data files to an instruction set

implementation.

- **Creating a Custom JSON Object Holder:** A quick demonstration of how to define a custom JSON object holder class.
- **Creating a Custom JSON Object Loader:** A quick demonstration of how to define a custom JSON loader class to work with an associated holder.
- **Registering the New Data File with an Instruction Set:** A quick demonstration of how to add the new JSON data file to the instruction set's registered files.
- **Calling the Static Main:** In this section we reviewed the different ways you can call the GenAsm assembler and the arguments that can be provided the program.
- **Breakdown of Output Files:** A detailed breakdown of the files generated by the GenAsm assembler.

At this point in the text, we've completed all of the review work necessary to really start diving into the project's code. In the upcoming chapters we'll review the source code in detail and explain how the Thumb-1 instruction set was implemented in the GenAsm assembler's extensible coding model.

# Chapter 9: The Utility and Helper Classes

In this chapter we're going to start our exploration of the GenAsm assembler's code base with a review of some utility, helper classes that are vitally important to the project. This will give us a chance to introduce our code review methodology and give you a little experience working with it. In this chapter we'll take a look at the following classes.

Class	Package
Logger.java	net.middlemind.GenAsm
Utils.java	net.middlemind.GenAsm
FileLoader.java	net.middlemind.GenAsm.FileIO
FileUnloader.java	net.middlemind.GenAsm.FileIO

Because these are higher level utility classes, they should be fairly direct in their implementation. In any case, while reviewing the code here keep in mind the functionality provided by these classes is available throughout the project and provides a sort of baseline functionality that classes can access.

# Class Review Template

Describing the functionality and use of a Java class can be difficult. There can be a lot of context and configuration involved with certain classes. We'll attempt to overcome this difficulty by reviewing classes in detail using the following template. To clarify the aspects of the classes clouded by context and configuration details we'll demonstrate how classes work whenever possible.

1. Static/Constants/Read-Only Class Members
2. Enumerations
3. Class Fields
4. Pertinent Method Outline/Class Headers
5. Support Method Details
6. Main Method Details
7. Demonstration

Not every class will have topics of discussion for each section of the class review template. In such cases we'll simply omit those sections without much mention. If you notice a section is missing, it's safe to assume that section doesn't apply to the current class' review.

## Class Review: Logger.java

In this section we'll review the `Logger` class so that you can get an idea of how the class review process works. Now, the classes we're reviewing here are utility classes so there will most likely be a fair number of static fields and methods. In future class reviews we'll deal with more standard classes that have fewer static members. The `Logger` class has only static members, so we'll do away with most of the class review steps and focus on the pertinent ones listed as follows.

1. Static/Constants/Read-Only Class Members
2. Pertinent Method Outline/Class Headers
3. Demonstration

In the demonstration section we'll attempt to show you how to use the class being reviewed so you can get an idea of how it works. And without further ado let's start our first, formal, Java class review.

## Static/Constants/Read-Only Class Members: **Logger.java**

The `Logger` class consists mainly of static class members, so we'll review the class' static fields first followed by the static methods.

***Listing 9-1. Logger.java Static/Constants/Read-Only Class Members 1***

```
1 public static boolean LOGGING_ON = true;
```

List of static fields belonging to the `Logger` class.

The `Logger` class' only static field is the `LOGGING_ON` Boolean which is defaulted to true. This field turns logging on or off for the class, shutting off all logging output if need be. In the next listing we'll take a look at a summary of the class' static methods before going into a detailed review of each method. Let's take a look.

***Listing 9-2. Logger.java Static/Constants/Read-Only Class Members 2***

```
1 public static void wrl(String s);
2 public static void wr(String s);
3 public static void wrlErr(String s);
```

List of static methods belonging to the `Logger` class.

The static methods, listed previously, are all there to support logging functionality in the code base. Let's look at each method in more detail.

***Listing 9-3. Logger.java Static/Constants/Read-Only Class Members 3***

```
01 public static void wrl(String s) {
```

```

02     if (LOGGING_ON) {
03         System.out.println(s);
04     }
05 }
06
07 public static void wr(String s) {
08     if (LOGGING_ON) {
09         System.out.print(s);
10    }
11 }
12
13 public static void wrErr(String s) {
14     if (LOGGING_ON) {
15         System.err.println(s);
16    }
17 }
```

Details of the static method belonging to the `Logger` class.

The first method shown in the previous listing is the `wrl` method. It takes a string argument and writes the string to standard output if the `LOGGING_ON` field is set to true. Notice that this method uses the `System.out.println` system method to actually print the string. The next method, `wr`, is similar but it doesn't use a print line method to write to standard output. The last method in the listing is the `wrlErr` method. This is another logging method, but it writes to `System.err` instead of `System.out`. In the next section we'll take a look at how the `Logger` class is defined.

## Pertinent Method Outline/Class Header: `Logger.java`

Because the `Logger` class is a static utility class comprised of mainly static members there isn't a pertinent class outline to speak of, so we'll omit that section. We'll move on to the next section and look at the `Logger` class' headers and declaration.

*Listing 9-4. `Logger.java` Pertinent Method Outline/Class Headers 1*

```
1 package net.middlemind.GenAsm;
```

```
2  
3 public class Logger { }
```

Details of the `Logger` class' declaration.

The `Logger` class declaration is very simple and direct. It is an example of a basic class declared in the `net.middlemind.GenAsm` package. There's not much to talk about here. Let's move on to the next section and take a look at the class in action.

## Demonstration: `Logger.java`

In this section we'll take a look at the `Logger` class in action. In the subsequent listing a series of error checks are performed, let's take a look.

*Listing 9-5. `Logger.java` Demonstration*

```
01 if (Utils.IsEmpty(ASM_SETS_FILE_NAME)) {  
02     Logger.wrlErr("GenAsm: Main: Error: No assembly source file  
provided.");  
03 } else if (Utils.IsEmpty(ASM_TARGET_SET)) {  
04     Logger.wrlErr("GenAsm: Main: Error: No assembly target set  
provided.");  
05 } else if (Utils.IsEmpty(ASM_SETS_LOADER_CLASS)) {  
06     Logger.wrlErr("GenAsm: Main: Error: No assembly set loader  
provided.");  
07 }  
08 }  
09  
10 }
```

A demonstration of the `Logger` class in use.

There's not much to review here. The `Logger` class is cross cutting, so it's very easy to use from anywhere in the project. Simply call one of the static logging methods as needed. The code in the previous listing is a demonstration of the `Logger` class' `wrlErr` method in use.

# Class Review: Utils.java

The `Utils` class is a centralized utility class that provides access to a number of helpful static methods. In fact, this class only has static methods, so we'll use only the following steps to review this class.

1. Static/Constants/Read-Only Class Members
2. Pertinent Method Outline/Class Headers
3. Demonstration

There are a number of useful formatting and conversion methods available in the `Utils` class. Let's me list the class methods here before we start the review process. I'll remove the redundant "public static" keywords for simplicity and readability but don't forget these are static methods we're reviewing.

*Listing 9-6. Utils.java Pertinent Static Method Outline 1*

```
//Unit Test Helper Methods
boolean CheckAssemblerTestProgramAgainstAnswers(JsonObjLineHexReps
hexDataLines, Hashtable<String, String> hashMap);

//File IO Methods
void WriteObject(Object obj, String name, String fileName, String
rootOutputDir);

void PrintObject(Object obj, String name);

//Encoding Adjustment Methods
String EndianFlipBin(String binStr);
String EndianFlipHex(String hexStr);
String ShiftBinStr(String binStr, int shiftCount, boolean shiftRight,
boolean padZeros);

//String Formatting Methods
String CleanHexPrefix(String hexStr);
String CleanBinPrefix(String binStr);
String SpaceString(String s, int len, boolean padLeft);
String FormatHexString(String s, int len);
```

```

String FormatHexString(String s, int len, boolean padLeft);
String FormatBinString(String s, int len);
String FormatBinString(String s, int len, boolean padLeft);
String FormatBinString(String s, int len, boolean padLeft, String
padWith);

String PrettyBin(String binStr, int len, boolean padLeft);
String PrettyHex(String hexStr, int len, boolean padLeft);

//Check Variable Methods
boolean IsStringEmpty(String s);
boolean IsListEmpty(List l);
int[] StringContainsArrayEntry(String[] target, String source);
boolean ArrayContainsString(String[] source, String target);
boolean ArrayContainsInt(int[] source, int target);

//Conversion Methods
byte[] ToBytes(int i);
Integer ParseNumberString(String s);
String Bin2Hex(String binStr);
String Hex2Bin(String hexStr);
int[] GetIntsFromRange(String range);
int[] GetIntsFromRange(String range, String rangeDelim);
int GetIntFromChar(char c);
int GetIntFromString(String c);

```

All entries in this listing are “public static” class methods. The extra key words have been omitted for redundancy.

Take a moment to look over the previously shown list of static class methods. Let your imagination run as you do so. See if you can figure out how the listed methods are used. In the next section we’ll review these static class methods in detail. Let’s jump into some code!

## **Static/Constants/Read-Only Class Members: Utils.java**

The `Utils` class has a few static class methods for us to review. We’ll do so by working through the different categories, written in the code comments, in

the previous listing. The first such category we have to review contains four methods that are used to help with the project's unit tests.

## Unit Test Helper Methods

The unit test helper methods are used by the project's unit tests by supporting a comparison of two hexadecimal listings. The method supports allowing a certain number of errors to occur. In this case an error is a mismatch. Due to the differences in the VASM, the assembler we use to check out assembler, there can be a few discrepancies in the hexadecimal listing.

*Listing 9-7. Utils.java Static Class Members - Unit Test Helper Methods 1*

```
01 public static boolean
CheckAssemblerTestProgramAgainstAnswers(JsonObjLineHexReps hexDataLines,
Hashtable<String, String> hashMap) {
02     int hmCnt = hashMap.size();
03     int errCnt = 0;
04     double prct = 0.0;
05     double prctMax = 0.10f;
06
07     for (String key : hashMap.keySet()) {
08         String val = hashMap.get(key);
09         key = key.toUpperCase();
10         val = val.toUpperCase();
11         boolean found = false;
12         for (JsonObjLineHexRep hexLine : hexDataLines.line_hex_reps) {
13             String aKey = hexLine.addressHex.toUpperCase();
14             String aVal = hexLine.valueHex.toUpperCase();
15             if (aKey.equals(key) && aVal.equals(val)) {
16                 found = true;
17                 break;
18             }
19         }
20
21         if (!found) {
22             Logger.wrl("Could not find a match for key: " + key + " with
value: " + val);
23             errCnt++;
24         }
25     }
```

```
26     prct = (errCnt / hmCnt);
27     if (prct > prctMax) {
28         Logger.wrl("Reached maximum allowed numbers of errors, " +
29         errCnt + ", with percentage, " + prct);
30         return false;
31     }
32     return true;
33 }
```

A listing of the, `CheckAssemblerTestProgramAgainstAnswers`, method used to help with the project's unit tests.

The `CheckAssemblerTestProgramAgainstAnswers` method is used to verify the results of running the assembler on a standard program source code and comparing the resulting binary data against the binary output of an existing assembler that supports the ARM Thumb-1 instruction set. As we've mentioned earlier, in our case we used the VASM assembler to generate the comparison binary output that is stored in JSON format.

Let's take a look at the method's arguments next. The first argument is an instance of the `JsonObjLineHexReps` class which, if you recall from previous chapters, is a data holder class that is most likely populated by a loader class. The second argument is a `Hashtable` that maps a string to a string. In this case the first string, the key, is a hexadecimal line address, and the second string is the hexadecimal representation of that assembly line. Both numbers are stored as a string representation of the hex number for simplicity and ease of use.

This method allows us to compare binary output from two different assemblers. Next up, we'll take a look at the method's code. The first few lines, 2 – 5, are used to prep local variables. The `hmCnt` variable keeps track of the number of entries in the `hashMap` argument. The `errCnt` variable, line 3, is used to track errors. An error in this case happens when the hexadecimal representation of an instruction at a specific memory address isn't the same in both lists.

Subsequently, we have the `prct` variable, which is used to track the current, calculated error percentage. The `prctMax` variable is used to hold a maximum allowed error percentage. In this case the maximum number of errors cannot exceed 10% of the total lines in the program. On line 7 we iterate over the `hashMap`'s keys. This gives us a unique set to iterate over that can be used to pull a value out of the `Hashtable`, lines 8 – 10. On line 11 we declare a local variable to track if we've found the match we're looking for.

For each key, value pair in the provided `hashMap` we check to see if there is a matching entry in the `hexDataLines` argument, lines 12 – 19. If one is found we exit the loop immediately, lines 15 – 18. If no matching assembly lines can be found, we log the error and increment the error count on lines 21 – 24. Lastly on lines 26 – 30 we recalculate the error percentage and compare it to the maximum allowed percentage. If the maximum allowed error is exceeded, we log the issue and return false, otherwise we return true. The next group of utility methods we're going to look at are the file IO methods.

## File IO Methods

The file IO methods are used to support simple reading and writing of files. They are mainly used to read and write JSON data files at different steps in the assembly process.

*Listing 9-8. Utils.java Static Class Members - File IO Methods 1*

```
1 public static void WriteObject(Object obj, String name, String fileName,
String rootOutputDir) throws IOException {
2     Logger.wrl("Utils: WriteObject: Name: " + name);
3     GsonBuilder builder = new GsonBuilder();
4     builder.setPrettyPrinting();
5     Gson gson = builder.create();
6     String jsonString = gson.toJson(obj);
7     FileUnloader.WriteString(Paths.get(rootOutputDir, fileName).toString(),
jsonString);
8 }

1 public static void PrintObject(Object obj, String name) {
2     Logger.wrl("Utils: PrintObject: Name: '" + name + "'');
```

```
3     GsonBuilder builder = new GsonBuilder();
4     builder.setPrettyPrinting();
5     Gson gson = builder.create();
6     String jsonString = gson.toJson(obj);
7     Logger.wr(jsonString);
8 }
```

The static file IO methods of the Utils class.

The file IO methods, listed previously, provide support for writing out objects to a file or to the standard output. The first method listed, `WriteObject`, takes four arguments. The first argument, `obj`, is the object to be written to an output file. The second argument is the name of the object, while the last two arguments specify the file name and the output directory of the target output file.

On line 2 we log the name of the object being written followed by prepping the JSON writer we're using to write the given object, lines 3 – 5. On line 6 a JSON representation of the `obj` object is created with a call to the `builder.create` method. The results are stored in the `jsonString` variable. Lastly, on line 7, the JSON object is written to a file with a call to the `FileUnloader` class' `WriteStr` method.

In the next method listed we have the `PrintObject` method. This method is similar to the one we've just reviewed save that it writes the object to standard output and not to a text file. On line 2 we log the name of the object we're printing out. The code on lines 3 – 6 is used to convert the given object, `obj`, to a JSON representation. The JSON string is then written to standard output on line 7. Next up, we'll look into the encoding adjustment helper methods.

## Encoding Adjustment Methods

The encoding adjustment methods are used to do just that, adjust a provided binary encoding. There are two forms of the method one that works with binary string representations and one that works with hexadecimal string representations. Let's take a look.

*Listing 9-9. Utils.java Static Class Members - Encoding Adjustment Methods 1*

```
01 public static String EndianFlipBin(String binStr) {  
02     binStr = CleanBinPrefix(binStr);  
03     int len = binStr.length();  
04     String[] bytes = new String[len / 8];  
05     int currentByteIdx = 0;  
06     String currentByte = "";  
07  
08     for (int i = 0; i < len; i++) {  
09         currentByte += binStr.charAt(i);  
10         if (currentByte.length() == 8) {  
11             bytes[currentByteIdx] = currentByte;  
12             currentByteIdx++;  
13             currentByte = "";  
14         }  
15     }  
16  
17     String res = "";  
18     for (int j = bytes.length - 1; j >= 0; j--) {  
19         res += bytes[j];  
20     }  
21     return res;  
22 }  
  
01 public static String EndianFlipHex(String hexStr) {  
02     hexStr = CleanHexPrefix(hexStr);  
03     int len = hexStr.length();  
04     String[] bytes = new String[len / 2];  
05     int currentByteIdx = 0;  
06     String currentByte = "";  
07  
08     for (int i = 0; i < len; i++) {  
09         currentByte += hexStr.charAt(i);  
10         if (currentByte.length() == 2) {  
11             bytes[currentByteIdx] = currentByte;  
12             currentByteIdx++;  
13             currentByte = "";  
14         }  
15     }  
16  
17     String res = "";  
18     for (int j = bytes.length - 1; j >= 0; j--) {
```

```
19         res += bytes[j];
20     }
21     return "0x" + res;
22 }
```

The first set of encoding adjustment methods from the `Utils` class.

The first method in the previous listing, `EndianFlipBin`, takes a string argument, `binStr`, that's a string representation of a binary number. Line 2 of the method is important because it uses another `Utils` class method, `CleanBinPrefix`, to make sure the binary string argument does not have any unnecessary prefix text. The GenAsm assembler supports a few different number formats, and some require a prefix string. For instance, a hexadecimal number has the prefix '0x', and binary numbers have the prefix '0b'. You can also use '#0b', '0b', '#0B', or '0B' to designate a binary number.

In order to interact with binary and hex strings we need to make sure they have been cleaned and don't have any prefix text. Local variables are initialized on lines 2 – 6. Note that the `bytes` variable is an array of strings. We'll use this array to break the binary string into 8-bit pieces. On lines 8 – 15 we separate the binary string as indicated previously. Then on lines 17 – 20 we reverse the order of the bytes, the endian flip, and finalize the return variable, `res`. Lastly, on line 21 we return the new binary string encoding.

The second encoding adjustment method for us to look at is the `EndianFlipHex` method. This method takes one argument, a string representation of a hexadecimal number, `hexStr`. On line 2 we clean the hex string argument by removing any prefix text from the string representation of the number. Notice that on line 4 we initialize the `bytes` array in much the same way we did in the previous method, except in this case we only divide by 2 instead of 8. The reason for this is that one byte in hex representation takes two characters instead of 8, as with binary representations.

The remainder of the method is just about identical to the binary version, so I won't go over it again in any detail. Note that the return value in this version of the method prepends the '0x' string to the returned hex value. The reason for this is that hexadecimal numbers are almost always prefixed with this string whereas binary strings don't really have a similar convention.

It's not so often that you see '0b' prefix a binary string. The last method for us to review in this group is the `ShiftBinStr` method shown in the subsequent listing.

***Listing 9-10. Utils.java Static Class Members - Encoding Adjustment Methods 2***

```
01 public static String ShiftBinStr(String binStr, int shiftCount, boolean
shiftRight, boolean padZeros) {
02     String binStr2 = binStr;
03     String p1 = null;
04     for (int i = 0; i < shiftCount; i++) {
05         p1 = null;
06         if (shiftRight) {
07             p1 = binStr2.substring(0, binStr2.length() - 1);
08             if (padZeros) {
09                 p1 = "0" + p1;
10             } else {
11                 p1 = "1" + p1;
12             }
13             binStr2 = p1;
14         } else {
15             p1 = binStr2.substring(1);
16             if (padZeros) {
17                 p1 += "0";
18             } else {
19                 p1 += "1";
20             }
21             binStr2 = p1;
22         }
23     }
24     return binStr2;
25 }
```

The second set of encoding adjustment methods from the `Utils` class.

The `ShiftBinStr` method is the last of the encoding adjustment methods for us to review. This method takes four arguments starting with the binary string to shift, `binStr`. The next argument is the `shiftCount`, or the number of positions to shift. The last two method arguments are Boolean flags. The first one, `shiftRight`, indicates which direction the shift should

be performed in while the last one, `padZeros`, indicates if zeros or ones should be used to pad the binary number when performing the shift operation.

Method variables are initialized on lines 2 – 3, while the main processing loop starts on line 4, and executes for the desired number of shift operations. If the shift operation is a shift right operation, lines 7 – 13, we remove the last character in the binary string and then prepend a ‘0’ or a ‘1’ on lines 8 – 12. The return value, `binStr2`, is updated on each iteration of the loop to reflect the current state of the shift operation.

In this case of a left shift, the code on lines 15 – 21 executes, removing the first character in the binary string and appending a zero or a one, lines 16 – 20, to the beginning of the string. The return variable, `binStr2`, is updated on lines 13 or 21 and returned from the method on line 24. That brings us to the conclusion of the encoding adjustment static method review. In the next section we’ll look into the class’ group of text formatting methods.

## Text Formatting Methods

The `Utils` class has a few text formatting methods for us to look at. These methods are used to format hexadecimal strings and binary strings as well as to provide a standard way to add spacing to a string.

*Listing 9-11. Utils.java Static Class Members - Text Formatting Methods 1*

```
01 public static String CleanHexPrefix(String hexStr) {  
02     return hexStr.replace("#0x", "").replace("0x", "").replace("#0X",  
"").replace("0X", "").replace("&", "");  
03 }  
  
01 public static String CleanBinPrefix(String binStr) {  
02     return binStr.replace("#0b", "").replace("0b", "").replace("#0B",  
"").replace("0B", "");  
03 }  
  
01 public static String SpaceString(String s, int len, boolean padLeft) {  
02     String ret = s;  
03     if (!IsEmpty(s)) {  
04         if (s.length() < len) {  
05             for (int i = s.length(); i < len; i++) {
```

```

06             if (padLeft) {
07                 ret = " " + ret;
08             } else {
09                 ret += " ";
10             }
11         }
12     } else if (s.length() > len) {
13         if (padLeft) {
14             ret = ret.substring(s.length() - len);
15         } else {
16             ret = ret.substring(0, (s.length() - len));
17         }
18     }
19 }
20 return ret;
21 }
```

The first set of text formatting methods from the Utils class.

In the first set of text formatting methods, shown in the previous listing, there are two very short methods `CleanHexPrefix` and `CleanBinPrefix` that are used to clean numeric prefixes from the string representation of the number. These methods are short and direct, be sure to look over them before moving on. The next method in the listing is the `SpaceString` method. This method is used to add spacing to the string argument provided, `s`. The next two method arguments, `len` and `padLeft`, indicate how much space to add and whether or not to add it to the left or right end of the string respectively.

Local method variables are initialized on line 2 and if the string argument is defined and it is shorter than the spacing number provided the code on lines 5 – 11 executes. In this section of code, padding is added to the left or the right of the string. If the string argument is longer than the spacing number, then the code on lines 13 – 17 executes. In this section of code extra characters are removed from the left- or right-hand side of the string. The final adjusted string is returned on line 20.

*Listing 9-12. Utils.java Static Class Members - Text Formatting Methods 2*

```

01 public static String FormatHexString(String s, int len) {
02     return FormatHexString(s, len, true);
03 }

04 public static String FormatHexString(String s, int len, boolean
05 padLeft) {
06     String ret = "";
07     if (!IsEmpty(s)) {
08         s = CleanHexPrefix(s);
09         ret = s;
10         if (s.length() < len) {
11             for (int i = s.length(); i < len; i++) {
12                 if (padLeft == true) {
13                     ret = "0" + ret;
14                 } else {
15                     ret += "0";
16                 }
17             }
18         }
19     }

```

The second set of text formatting methods from the Utils class.

In the next set of string formatting methods for us to look at we have a set of methods used to adjust the text of a hex string. The first method listed is an overloaded pass through method. This method simply calls the overloaded version of the method and supplies some default values for the overloaded method's arguments, line 2. Moving on to look at the overloaded version of the method, this method takes three arguments. The first is the hex string to format, `s`, while the second and third arguments indicate the length of the padding and the side to add the padding to respectively.

Method variables are defined on line 2, and the string argument is validated on line 3. If the string argument `s` is defined, then the code on lines 4 – 16 executes. The hexadecimal number prefix is removed on line 4 and the return variable is updated on line 5. If the string argument needs padding, line

6, then the padding is added to the left- or right-hand side of the string on lines 7 – 13. On line 15 the hex string is converted to uppercase, to follow hex number display conventions, and a prefix is added to the return string on line 16. Lastly, the adjusted string, if any, is returned on line 18.

*Listing 9-13. Utils.java Static Class Members - Text Formatting Methods 3*

```
01 public static String FormatBinString(String s, int len) {  
02     return FormatBinString(s, len, true);  
03 }  
  
01 public static String FormatBinString(String s, int len, boolean  
padLeft) {  
02     return FormatBinString(s, len, true, "0");  
03 }  
  
01 public static String FormatBinString(String s, int len, boolean  
padLeft, String padWith) {  
02     String ret = null;  
03     if (!IsEmpty(s)) {  
04         s = CleanBinPrefix(s);  
05         ret = s;  
06         if (s.length() < len) {  
07             for (int i = s.length(); i < len; i++) {  
08                 if (padLeft) {  
09                     ret = padWith + ret;  
10                 } else {  
11                     ret += padWith;  
12                 }  
13             }  
14         } else if (s.length() > len) {  
15             if (padLeft) {  
16                 ret = ret.substring(s.length() - len);  
17             } else {  
18                 ret = ret.substring(0, (s.length() - len));  
19             }  
20         }  
21     }  
22     return ret;  
23 }
```

The third set of text formatting methods from the Utils class.

The third set of text formatting methods is used to adjust the format of string representations of binary numbers. The first two method entries are very simple. They are overloaded pass through methods. Each one calls a more complex version of the method providing a default value for any new method arguments. Take a moment to look over these method entries and understand how they work. Also, note the flexibility in calling the method provided by the overloaded versions of the method. Let's move on to look at the main version of the method, the one that takes four arguments.

The first method argument, `s`, is the string representing a binary number that we need to format. The second argument specifies the desired length of the string, while the third and fourth arguments specify the side of the string to add padding to and the string to use as padding, respectively. The method return variable, `ret`, is declared on line 2, and if the string argument is not empty then the code on lines 4 – 20 executes. If not, the `ret` variable is returned, line 22, causing the method to return null. On lines 4 – 5 the string argument is cleaned of any prefix text and the local return variable is updated. If the string requires padding the code in lines 7 – 13 executes. If the string requires text to be removed, then the code on lines 15 – 19 executes.

***Listing 9-14. Utils.java Static Class Members - Text Formatting Methods 4***

```
01 public static String PrettyBin(String binStr, int len, boolean padLeft)
{
02     binStr = CleanBinPrefix(binStr);
03     String s = FormatBinString(binStr, len, true);
04     String ret = "";
05     int l = s.length();
06     String tmp = "";
07     for (int i = 0; i < l; i++) {
08         tmp += s.charAt(i);
09         if (i > 0 && (i + 1) % 8 == 0) {
10             ret += tmp;
11             tmp = "";
12             if (i < l - 1) {
13                 ret += " ";
14             }
15         }
16     }
17     if (padLeft) {
18         for (int i = 0; i < len - l; i++) {
19             ret = " " + ret;
20         }
21     }
22     return ret;
23 }
```

```

15      }
16  }
17  return ret;
18 }

01 public static String PrettyHex(String hexStr, int len, boolean padLeft)
{
02     hexStr = CleanHexPrefix(hexStr);
03     String s = FormatHexString(hexStr, len, padLeft);
04     s = CleanHexPrefix(s);
05     String ret = "";
06     int l = s.length();
07     String tmp = "";
08     for (int i = 0; i < l; i++) {
09         tmp += s.charAt(i);
10         if (i > 0 && i % 2 == 1) {
11             ret += tmp;
12             tmp = "";
13             if (i < l - 1) {
14                 ret += " ";
15             }
16         }
17     }
18     return "0x" + ret;
19 }

```

The fourth set of text formatting methods from the Utils class.

The last set of text formatting methods for us to review is used to pretty up a string representation of a number by adding a space at the correct interval given the number base and method version, binary or hex. The first method, `PrettyBin`, takes three arguments. The first argument is the string to process. The second indicates the desired string length, while the third argument specifies if the padding should be added to the left- or right-hand side of the provided string.

The method's local variables are initialized on lines 2 – 6. The string argument, `s`, is cleaned and formatted on lines 2 – 3. This results in the string that will be processed by the method's remaining code. The loop on lines 7 – 16 scans through the new string representation of a binary number and finds

the byte boundaries, or the bit locations that are divisible by eight. At the end of the method the newly formatted string is returned, line 17.

The hexadecimal version of this method, `PrettyHex`, is very similar so we'll only talk about the main differences. Notice that on line 10 the spacing test is now set to find byte boundaries of hexadecimal numbers, or every two characters. Also note that the string cleaning method calls are all geared to hexadecimal numbers. The remaining difference with this pretty formatting method is that it adds a standard hexadecimal prefix to the returned string.

Note that the method cleans the input argument so it can work with a string resulting from a call to itself. It's important to always handle prefixes when dealing with string representations of binary and hexadecimal numbers. Next up, we'll look at the methods that are used in variable validation.

## Validation Methods

The validation methods are used to simplify a few common validation tasks in the project. This includes empty checks and a method for finding matches in an array.

*Listing 9-15. Utils.java Static Class Members - Validation Methods 1*

```
01 public static boolean IsStringEmpty(String s) {  
02     if (s == null || s.equals("")) {  
03         return true;  
04     } else {  
05         return false;  
06     }  
07 }  
  
01 public static boolean IsListEmpty(List l) {  
02     if (l == null) {  
03         return true;  
04     } else if (l != null && l.isEmpty()) {  
05         return true;  
06     } else {  
07         return false;  
08     }  
09 }
```

```

01 public static int[] StringContainsArrayEntry(String[] target, String
source) {
02     int idx = -1;
03     for (int i = 0; i < target.length; i++) {
04         idx = source.indexOf(target[i]);
05         if (idx != -1) {
06             return new int[] { i, idx };
07         }
08     }
09     return null;
10 }
```

The first set of validation methods from the Utils class.

The first method in this set is the `IsEmpty` method. This method is simple and direct. It provides a standard way to check for emptiness in string variables. The benefit here is that we've centralized the check. This will greatly simplify string validation throughout the project. The second method, `IsEmpty`, shown in the previous listing is similar but, in this case, we're checking a data structure for emptiness. Similarly, though there is more than one case that can result in an empty list object. Note that if the list is null or if it is defined and contains no child objects the method returns true, indicating the list is empty.

The next method in this set is the `StringContainsArrayEntry` method and it's used to check if a string, `source`, contains a sub-string that is an element of the array argument, `target`. Notice that this method returns an array of integers. The integers in this array are the index of the matching array element and the position in the `source` string that the array element occurs. If nothing is found the method returns null. Let's take a look at the second set of variable validation methods.

#### ***Listing 9-16. Utils.java Static Class Members - Validation Methods 2***

```

1 public static boolean ArrayContainsString(String[] source, String
target) {
2     for (int i = 0; i < source.length; i++) {
3         if (source[i].equals(target)) {
4             return true;
```

```

5      }
6  }
7  return false;
8 }

1 public static boolean ArrayContainsInt(int[] source, int target) {
2   for (int i = 0; i < source.length; i++) {
3     if (source[i] == target) {
4       return true;
5     }
6   }
7   return false;
8 }

```

The second set of validation methods from the Utils class.

In the next method for us to look at, `ArrayContainsString`, an array argument, `source`, is searched for an entry with the value of the `target` argument. This version of the method is designed to work with strings. Another common type of array is an integer array. There is a utility method for searching through integer arrays called `ArrayContainsInt` that is similar to the method we've just reviewed. The only difference in this case is that the method is designed to work with integers instead of strings.

These methods are very simple so I'll leave their review to you. The last section for us to look at is the conversion methods section. This group of utility methods is used when converting and manipulating different data types.

## Conversion Methods

The conversion methods group has a few different methods used to convert between hexadecimal, binary, byte, and integer numeric representations.

***Listing 9-17. Utils.java Static Class Members - Conversion Methods 1***

```

01 public static byte[] ToBytes(int i) {
02   byte[] result = new byte[4];
03   result[0] = (byte)(i >> 24);
04   result[1] = (byte)(i >> 16);

```

```

05     result[2] = (byte)(i >> 8);
06     result[3] = (byte)(i /*>> 0*/);
07     return result;
08 }

01 public static Integer ParseNumberString(String s) {
02     Integer tInt = null;
03     s = s.trim();
04     if (s.contains("#0x")) {
05         tInt = Integer.parseInt(s.replace("#0x", ""), 16);
06     } else if (s.contains("0x")) {
07         tInt = Integer.parseInt(s.replace("0x", ""), 16);
08     } else if (s.contains("&")) {
09         tInt = Integer.parseInt(s.replace("&", ""), 16);
10    } else if (s.contains("#0b")) {
11        tInt = Integer.parseInt(s.replace("#0b", ""), 2);
12    } else if (s.contains("0b")) {
13        tInt = Integer.parseInt(s.replace("0b", ""), 2);
14    } else if (s.contains("#")) {
15        tInt = Integer.parseInt(s.replace("#", ""), 16);
16    } else {
17        tInt = Integer.parseInt(s, 10);
18    }
19    return tInt;
20 }

01 public static String Bin2Hex(String binStr) {
02     String s = binStr;
03     s = CleanBinPrefix(s);
04     s = "#0b" + s;
05     Integer v = Utils.ParseNumberString(s);
06     return Integer.toHexString(v);
07 }

01 public static String Hex2Bin(String hexStr) {
02     String s = hexStr;
03     s = CleanHexPrefix(s);
04     s = "#0x" + s;
05     Integer v = Utils.ParseNumberString(s);
06     return Integer.toBinaryString(v);
07 }

```

The first set of conversion methods from the Utils class.

In the first group of conversion methods for us to review we have a couple of very useful utility methods. The first such method, `ToBytes`, takes an integer argument and using bit shifting techniques each byte in the big-endian binary representation of the integer is extracted and stored in an array, lines 3 – 6. The array is returned on line 7 of the method.

The next method for us to look at is the `ParseNumberString` method. This method is used to parse the assembly source string of a number token. In other words, after the lexer runs and the line of source code has been broken into string segments the tokenizer then runs and identifies each string segment as a certain token type. A token can be of any type defined by the `is_entry_types.json` data file. Think back to our review of the data files and how they are used to drive the assembler's Thumb-1 implementation.

In order to extract the real numeric value of a number expressed in assembly language source code we need to handle certain number prefixes. We've encountered this before when reviewing the text formatting methods. Take a closer look at this method. Notice that there are three ways to write a hexadecimal number, two ways to write a decimal number, and two ways to write a binary number. Make sure you are familiar with the prefixes used for the different numeric bases. I'll summarize them here.

<b>Number Type</b>	<b>Prefix</b>
Hexadecimal	#0x
Hexadecimal	0x
Hexadecimal	&
Decimal	#
Decimal	No prefix
Binary	#0b
Binary	0b

The last two methods in this set are two sides of the same coin. The `Bin2Hex` and `Hex2Bin` methods are convenient utility methods for converting between binary and hexadecimal string representations of numeric

values. Let's review the `Bin2Hex` method first. The local variable `s` is initialized on line 2 and on lines 3 – 4 the binary prefix, if any, is normalized. The number is parsed on line 5 and returned on line 6 in hexadecimal form. The `Hex2Bin` method is very similar except that it is designed to normalize a hexadecimal number prefix and return a binary representation of the number value. Let's look at the next, and last, set of static methods for us to review.

***Listing 9-18. Utils.java Static Class Members - Conversion Methods 2***

```
01 public static int[] GetIntsFromRange(String range) throws
ExceptionMalformedRange {
02     return GetIntsFromRange(range, JsonObjTxtMatch.special_range);
03 }

01 public static int[] GetIntsFromRange(String range, String rangeDelim)
throws ExceptionMalformedRange {
02     int[] ret = new int[2];
03     String[] strInts = range.split(rangeDelim);
04     if (strInts.length == 2) {
05         try {
06             ret[0] = Integer.parseInt(strInts[0]);
07             ret[1] = Integer.parseInt(strInts[1]);
08         } catch (NumberFormatException e) {
09             throw new ExceptionMalformedRange("The range string provided
is not properly formed, " + range + ", with delimiter '" + rangeDelim +
""");
10         }
11
12         if (ret[0] > ret[1]) {
13             throw new ExceptionMalformedRange("The range string provided
is not properly formed, " + range + ", with delimiter '" + rangeDelim +
""");
14         }
15     } else {
16         throw new ExceptionMalformedRange("The range string provided is
not properly formed, " + range + ", with delimiter '" + rangeDelim + "'");
17     }
18     return ret;
19 }

01 public static int GetIntFromChar(char c) throws ExceptionMalformedRange
{
```

```

02     int ret = 0;
03     try {
04         ret = Integer.parseInt(c + "");
05     } catch (NumberFormatException e) {
06         throw new ExceptionMalformedRange("The character provided could
07         not be converted to a digit, " + c);
08     }
09     return ret;
10 }

11 public static int GetIntFromString(String c) throws
ExceptionMalformedRange {
12     int ret = 0;
13     try {
14         ret = Integer.parseInt(c + "");
15     } catch (NumberFormatException e) {
16         throw new ExceptionMalformedRange("The character provided could
17         not be converted to a digit, " + c);
18     }
19     return ret;
20 }
```

The second set of conversion methods from the Utils class.

The second, and last, set of conversion methods are shown in the previous listing. The first method, `GetIntsFromRange`, is a pass through to an overloaded version of this method. This method provides a default value for the range delimiter. The overloaded version of the `GetIntsFromRange` method takes two arguments. A range to process, in string form, and a range delimiter, `rangeDelim`. Notice that this pair of methods throws an exception if there is an issue with how the range is expressed.

Method variables are initialized on lines 2 – 3. If after splitting the range using the provided range delimiter there are two objects found, we attempt to convert them to integers on lines 5 – 10. A check is performed on lines 12 – 14 to see if the range entries are in ascending order. In either case, during the integer conversion or the order test, if an issue is encountered the method throws an `ExceptionMalformedRange` exception with a description of the issue found.

On lines 15 – 17, if there weren't two objects found after splitting the range, we also throw an exception. Lastly on line 18 we return the `ret` variable which has the start and stop integer values from the range in array index 0 and 1 respectively. In the next conversion method, `GetIntFromChar`, a character is converted to an integer using the `Integer` class' `parseInt` method. If anything goes wrong an exception is thrown indicating what the issue encountered was.

The last method in this set, `GetIntFromString`, is very similar to the method we just reviewed. It can be used to convert a string to an integer, and it also throws an exception if any issues are encountered. That brings us to the conclusion of the static members section of the `Utils` class' review. In the next section we'll take a look at how the class is declared.

## Pertinent Method Outline/Class Header: Utils.java

Because the `Utils` class is a static helper class there are only static class members. As such the class won't have any entries in the pertinent method outline and since we've reviewed all class methods during the static members review section, we'll focus solely on the class' declaration.

*Listing 9-19. Utils.java Pertinent Method Outline/Class Headers 1*

```
01 package net.middlemind.GenAsm;
02
03 import com.google.gson.Gson;
04 import com.google.gson.GsonBuilder;
05 import java.io.IOException;
06 import java.nio.file.Paths;
07 import java.util.Hashtable;
08 import java.util.List;
09 import net.middlemind.GenAsm.Exceptions.Thumb.ExceptionMalformedRange;
10 import net.middlemind.GenAsm.FileIO.FileUnloader;
11 import net.middlemind.GenAsm.JsonObjs.JsonObjLineHexRep;
12 import net.middlemind.GenAsm.JsonObjs.JsonObjLineHexReps;
13 import net.middlemind.GenAsm.JsonObjs.JsonObjTxtMatch;
14
15 public class Utils { }
```

Take a quick look at the `Utils` class' declaration. Does anything stand out to you? One thing that should stand out is that our project is structured using packages to hold major aspects of the code base. For instance, loading classes, holding classes, and exceptions each have their own package in the project. Another thing that might stand out is that we're using `Gson` to handle processing JSON objects in Java. That's really all we have to cover here. In the next section we'll look at a demonstration of the `Utils` class in action.

## Demonstration: Utils.java

The `Utils` class is very broad in use and many of its methods are ubiquitous throughout the project's code base. I don't want to spend too much more time on this class, so we'll wrap things up with a short and sweet example demonstrating the use of an important conversion method.

*Listing 9-20. Utils.java Demonstration*

File: `Utils.java`

Method: `PopulateDirectiveArgAndAreaData`

Snippet:

```
1 } else if (foundOrg == true) {  
2     asmStartLineNumber = Utils.ParseNumberString(token.source);  
3     foundOrg = false;  
4 }
```

A demonstration of the `Utils` class in use.

The demonstration I chose is short but it's also a very powerful example that you can visualize easily. If you recall from our review of the GenAsm assembler's directives there was one entry called `@ORG` that's used to set the starting memory address for the current program. The code shown in the previous listing is an example of using the `Utils` class' `ParseNumberString` method to parse the assembler source code, `token.source`, and store the resulting integer value in the `Assembler`

class' `asmStartLineNumber` field. The code, lines 1 – 4, is controlled by a local variable, `foundOrg`, that tracks if the `@ORG` directive is encountered. That brings us to the conclusion of this review section. In the next section we'll take a look at the `FileLoader` class.

## Class Review: FileLoader.java

The next class up for review is the file IO helper class, `FileLoader`. This class is relatively concise, so we'll diverge from the review template we discussed earlier in the chapter and just display the entire class. In the subsequent listing the entire `FileLoader` class is shown. Please take a moment to look over the class before we review it.

*Listing 9-21. FileLoader.java Full Class Review*

```
01 package net.middlemind.GenAsm.FileIO;
02
03 import java.io.IOException;
04 import java.nio.file.Files;
05 import java.nio.file.Paths;
06 import java.util.List;
07
08 public class FileLoader {
09     public static List<String> Load(String file) throws IOException {
10         return Files.readAllLines(Paths.get(file));
11     }
12
13     public static String LoadStr(String file) throws IOException {
14         return Files.readString(Paths.get(file));
15     }
16
17     public static byte[] LoadBin(String file) throws IOException {
18         return Files.readAllBytes(Paths.get(file));
19     }
20 }
```

A complete listing of the `FileLoader` class.

The `FileLoader` class is another static helper class not dissimilar from the classes we've already reviewed in this chapter. The class uses the `java.io` and `java.nio` base packages to expose three different file loading methods. The first such method, `Load`, is designed to load a list of strings from a specified file. The second method shown, `LoadStr`, is similar but instead of loading a list of strings the method reads the entire contents of a specified file into one large string. The last static method exposed by the `FileLoader` class is the `LoadBin` method. This method is used for loading binary files and is responsible for reading the entire contents of the file into an array of bytes. In the next section we'll see how the class is used.

## Demonstration: `FileLoader.java`

The `FileLoader` class is used mainly as a way to load data files into the assembler to support a particular instruction set. We'll take a look at a few examples of the class in use in the subsequent listing.

*Listing 9-22. `FileLoader.java` Demonstration*

File: `PreProcessorThumb.java`

Method: `RunPreProcessor`

Snippet:

```
1 List<String> ret = FileLoader.Load(assemblySourceFile);
```

File: `PreProcessorThumb.java`

Method: `RunPreProcessor`

Snippet:

```
1 incBin = FileLoader.LoadBin(fileName);
2 numBytes = incBin.length;
3 numHalfWords = numBytes / 2;
```

File: `AssemblerThumb.java`

Method: `LoadAndParseJsonObjData`

Snippet:

```
1 json = FileLoader.LoadStr(entry.path);
```

A demonstration listing showing the `FileLoader` class in use.

Two of the examples shown in the previous listing are from the `PreProcessorThumb.java` file's `RunPreProcessor` method. In the first example we're loading the assembly source code file into a list of strings, so we use the `FileLoader` class' `Load` method. Notice how quick and easy it is to use. In the next example, also from the `RunPreProcessor` method, the example shows the class loading a binary file. The last example is from the `AssemblerThumb.java` file's `LoadAndParseJsonObjData` method. In this example we're loading the entire file's contents into one large string. This approach is used to load JSON data files for parsing. That takes care of this class review. In the next section we'll take a look at an example of the `FileUnloader` class in action.

## Class Review: `FileUnloader.java`

The `FileUnloader` class is similar to the `FileLoader` class we've just reviewed, save that it's responsible for providing methods that write data to a file.

*Listing 9-23. `FileUnloader.java` Full Class Review*

```
01 package net.middlemind.GenAsm.FileIO;
02
03 import java.io.BufferedReader;
04 import java.io.File;
05 import java.io.IOException;
06 import java.nio.file.Files;
07 import java.nio.file.Paths;
08 import java.nio.file.StandardOpenOption;
09 import java.util.List;
10
11 public class FileUnloader {
12     public static void WriteStr(String file, String str) throws
13         IOException {
14         File directory = new File(file);
15         directory = new File(directory.getParent());
16         if (! directory.exists()){
17             directory.mkdirs();
18         }
19         try {
20             Files.write(directory.toPath(), str.getBytes("UTF-8"),
21                         StandardOpenOption.CREATE);
22         } catch (IOException e) {
23             e.printStackTrace();
24         }
25     }
26 }
```

```
18
19     BufferedWriter bf = Files.newBufferedWriter(Paths.get(file),
20     StandardOpenOption.CREATE, StandardOpenOption.WRITE,
21     StandardOpenOption.TRUNCATE_EXISTING);
22     bf.write(str);
23     bf.flush();
24     bf.close();
25 }
26
27 public static void WriteList(String file, List<String> strs) throws
28 IOException {
29     File directory = new File(file);
30     directory = new File(directory.getParent());
31     if (! directory.exists()){
32         directory.mkdirs();
33     }
34
35     BufferedWriter bf = Files.newBufferedWriter(Paths.get(file),
36     StandardOpenOption.CREATE, StandardOpenOption.WRITE,
37     StandardOpenOption.TRUNCATE_EXISTING);
38     for(String s : strs) {
39         bf.write(s + System.lineSeparator());
40     }
41     bf.flush();
42     bf.close();
43 }
44
45 public static void WriteBuffer(String file, byte[] buff) throws
46 IOException {
47     File directory = new File(file);
48     directory = new File(directory.getParent());
49     if (! directory.exists()){
50         directory.mkdirs();
51     }
52
53     Files.write(Paths.get(file), buff);
54 }
```

A complete listing of the FileUnloader class.

The `FileUnloader` class, not unlike its file loading counterpart, uses a number of classes from the `java.io` and `java.nio` base packages. This class has methods that are the reverse of the methods provided by the `FileLoader` class. For instance, the `WriteStr` method writes a single string to a file. The `WriteList` method is used to write a series of strings to a file and lastly the `WriteBuffer` method is used to write a byte array to a file. This class is short and sweet and really powerful. In the next section we'll take a look at how the `FileUnloader` class is used.

## Demonstration: FileUnloader.java

The `FileUnloader` class is used to write data to files and provides some quick access, static methods, to do the job. In the next listing we'll take a look at some code demonstrating its use. Take a moment to note that the class tries to create the directory structure of the output file if it doesn't exist. Let's jump into some code!

*Listing 9-24. FileUnloader.java Demonstration*

File: Utils.java

Method: WriteObject

Snippet:

```
1 FileUnloader.WriteString(Paths.get(rootOutputDir, fileName).toString(),  
jsonString);
```

File: LinkerThumb.java

Method: RunLinker

Snippet:

```
1 if(!quellFileOutput) {  
2     FileUnloader.WriteLine(Paths.get(outputDir,  
"output_assembly_listing_endian_lil.list").toString(), 1stFile);  
3     FileUnloader.WriteBuffer(Paths.get(outputDir,  
"output_assembly_listing_endian_lil.bin").toString(), data);  
4 }
```

A demonstration listing showing the `FileUnloader` class in use.

In the demonstrations shown in the previous listing we have two

examples to look at. The first is from the `Utils` class' `WriteObject` method. The `WriteStr` method is used to write a string, usually a JSON string, to a file. The second example is from the `LinkerThumb` class' `RunLinker` method. In the example shown the `WriteList` method is used to write a list to a file while the `WriteBuffer` method writes binary data to a file. That brings us to the end of this code review chapter. Let's summarize what we've covered before moving on to the next chapter.

## Chapter Conclusion

In this chapter, we were introduced to the class review template. This template defines the general structure of a class' code review. We also got to take a moment and apply the code review template to the static helper classes, `Logger` and `Utils`. We also reviewed two rather concise classes, `FileLoader` and `FileUnloader`, and in doing so didn't use the class review template at all. In some cases, it doesn't make sense to use the more structured and complex review procedure if the class doesn't warrant it. The classes we reviewed in this chapter are mostly utility and support classes, summarized as follows.

- **Logger.java:** Provides convenient static method to help with logging throughout the project.
- **Utils.java:** A class that provides quick access, static methods, that help with a variety of different tasks from number conversions to text formatting, etc.
- **FileLoader.java:** A file IO class that is used to read different data types from a target file.
- **FileUnloader.java:** A file IO class that is used to write different data types to a target file.

Next up, we'll dive further into the project's code as we take a moment to review the base classes used by the GenAsm assembler project. These classes provide a foundation for an instruction set's actual implementation, that we'll review a little later on in this text.

# Chapter 10: The Base Classes Part 1

In this chapter we're going to start our exploration of the GenAsm assembler's core code base with a review of some classes and interfaces that are vitally important to the project. This will give us chance to use our code review methodology and give you a little more experience working with it. In this chapter we'll take a look at the following classes.

*Listing 10-1. The GenAsm Project's Base Classes*

Class	Package	Type
Assembler.java	net.middlemind.GenAsm.Assemblers	Interface
AssemblerEventHandler.java	net.middlemind.GenAsm.Assemblers	Interface
Symbols.java	net.middlemind.GenAsm.Assemblers	Class
Symbol.java	net.middlemind.GenAsm.Assemblers	Class
ExceptionBase.java	net.middlemind.GenAsm.Exceptions	Class
JsonObj.java	net.middlemind.GenAsm.JsonObjs	Interface
JsonObjBase.java	Net.middlemind.GenAsm.JsonObjs	Class
JsonObjBitRep.java	net.middlemind.GenAsm.JsonObjs	Class
JsonObjBitSeries.java	net.middlemind.GenAsm.JsonObjs	Class

JsonObjBitShift.java	net.middlemind.GenAsm.JsonObjs	Class
JsonObjIsSets.java	net.middlemind.GenAsm.JsonObjs	Class
JsonObjIsSet.java	net.middlemind.GenAsm.JsonObjs	Class
JsonObjIsFile.java	net.middlemind.GenAsm.JsonObjs	Class
JsonObjLineHexReps.java	net.middlemind.GenAsm.JsonObjs	Class
JsonObjLineHexRep.java	net.middlemind.GenAsm.JsonObjs	Class
JsonObjNumRange.java	net.middlemind.GenAsm.JsonObjs	Class
JsonObjTxtMatch.java	net.middlemind.GenAsm.JsonObjs	Class
Artifact.java	net.middlemind.GenAsm.Lexers	Class
ArtifactLine.java	net.middlemind.GenAsm.Lexers	Class
Lexer.java	net.middlemind.GenAsm.Lexers	Interface
Linker.java	net.middlemind.GenAsm.Linkers	Interface
Loader.java	net.middlemind.GenAsm.Loaders	Interface
LoaderBitSeries.java	net.middlemind.GenAsm.Loaders	Class
LoaderIsSets.java	net.middlemind.GenAsm.Loaders	Class
LoaderLineHexReps.java	net.middlemind.GenAsm.Loaders	Class
PreProcessor.java	net.middlemind.GenAsm.PreProcessor	Interface
Token.java	net.middlemind.GenAsm.Tokeners	Class
TokenLine.java	net.middlemind.GenAsm.Tokeners	Class
TokenSorter.java	net.middlemind.GenAsm.Tokeners	Class
Tonkener.java	net.middlemind.GenAsm.Tokeners	Interface

List of the GenAsm project's base classes.

I know it looks like a lot of work to try to accomplish in one chapter, but you'll soon see that many of the classes in the previous list are concise and will not take long to review. There are also a fair number of Java Interfaces, these entries will be exceptionally concise. Some listings will only have one line of code, the interface definition, and we can review them very quickly.

The classes in the previous list are considered base classes as many of them are the basis for instruction set specific classes in the project. The remaining classes are also considered base classes because their functionality is useful outside of a specific instruction set implementation. In our case, ARM Thumb-1. In the next section we'll begin the code review with the first set of classes that belong to the `net.middlemind.GenAsm.Assemblers` package.

## Package Review: Assemblers

The first package we'll look into is the GenAsm project's Assemblers package. When I refer to a package name I will omit the root of the package name, `net.middlemind.GenAsm`, as it is redundant from this point on. The assembler package contains base classes and interfaces that support running the assembler. These classes could be used for many different instruction set implementations so they have been placed in a general package. Let's take a look.

## Interface Review: Assembler.java

The `Assembler` interface defines the minimum set of methods necessary to define an assembler class that can plug into the GenAsm execution process. There will be more on this to come. Due to the concise nature of the class, we'll list it in its entirety here.

*Listing 10-2. Assembler.java Full Interface Review*

```
1 package net.middlemind.GenAsm.Assemblers;
2
3 import java.util.List;
4 import net.middlemind.GenAsm.JsonObjs.JsonObjIsSet;
5
6 public interface Assembler {
7     public void RunAssembler(JsonObjIsSet jsonIsSet, String
assemblySourceFile, List<String> assemblySource, String outputDir, Object
otherObj, AssemblerEventHandler asmEventHandler, boolean verbose, boolean
quellOutput) throws Exception;
8 }
```

A complete listing of the `Assembler` interface.

The `Assembler` interface is short and sweet. This interface describes the minimum implementation necessary to create an `Assembler` class implementation that plugs into the GenAsm execution process. One main thing to notice is that this interface uses the `JsonObjIsSet` class. That means that the `JsonObjIsSet` class is outside of a specific instruction set implementation. If you think back to our review of the JSON data files we said that the `is_sets.json` file was used to define all the instruction sets, their data files, and the classes used to load and hold them.

That's why the first argument to the `RunAssembler` method, `jsonIsSet`, is an instance of the `JsonObjIsSet` class. An assembler implementation needs access to any data files associated with its instruction set. The second argument, `assemblySourceFile`, is as you might expect, the path to the assembly file to process. The next argument, `assemblySource`, is a `List` of strings that represent the lines of assembly source code loaded from the specified assembly source code file, usually after the preprocessor is run.

The following argument, `outputDir`, is used to specify the output directory where the resulting binary output, listing, and report files, will be written. Next up, the `otherObj` argument is a generic Java object that can be used to customize the assembly process by providing a special class as an argument to an assembler implementation. In order to process events from the assembly process, allowing for further customization, an event handler class can be specified with the `asmEventHandler` argument.

The last two arguments are used to modify the behavior of the assembly process by providing Boolean flags to turn on verbose output and preventing output files from being written. Now that we've covered all the arguments outlined in the `RunAssembler` method I should mention that none of them mean anything unless they are implemented by the assembler class designed to support a target instruction set.

In our case the `AssmeblerThumb` class implements the `Assembler` interface and is designed to support the ARM Thumb-1 instruction set. Coming up next we'll see how the `Assembler` class is used.

## Demonstration: Assembler.java

The best way to demonstrate the `Assembler` interface in use is to look at a class that implements the interface. In the next listing we'll take a look at a few select lines of code. We'll review the `AssemblerThumb` class in detail a little later on in the text it wouldn't hurt to look at it now as a part of the demonstration.

*Listing 10-3. Assembler.java Demonstration*

File: `AssemblerThumb.java`

Method: n/a

Snippet:

```
01 public class AssemblerThumb implements Assembler { }
```

File: `AssemblerThumb.java`

Method: `RunAssembler`

Snippet:

```
01 public void RunAssembler(JsonObjIsSet jsonIsSet, String
assemblySourceFile, List<String> assemblySource, String outputDir, Object
otherObj, AssemblerEventHandler asmEventHandler, boolean verbose, boolean
quellOutput) throws Exception {
02     try {
03         ...
04     } catch(Exception e) {
05         Logger.wrlErr("Error in RunAssembler method on step: " +
lastStep);
06         if(lastLine != null) {
07             Logger.wrlErr("Last line processed: " + lastLine.lineNumAbs);
08             if(lastLine.source != null) {
09                 Logger.wrlErr("Last line source: " +
lastLine.source.source);
10             }
11         } else {
12             Logger.wrlErr("Last line processed: unknown");
13             Logger.wrlErr("Last line source: unknown");
14         }
15     }
16     if(lastToken != null) {
```

```

17     Logger.wrlErr("Last token processed: " + lastToken.source + "
with index " + lastToken.index + ", type name '" + lastToken.type_name +
"', and line number " + lastToken.lineNumAbs);
18 } else {
19     Logger.wrlErr("Last token processed: unknown");
20 }
21 throw e;
22 }
```

A demonstration of the Assembler interface in use.

The first example we'll look at is from the `AssemblerThumb` class' declaration. As shown the class implements the `Assembler` interface. The next example shows the class' definition of the `RunAssembler` method with the main body of code removed. Notice the definition of the `RunAssembler` with method signature matching the interface. Also note that although the main body of code is missing we can see how the `AssemblerThumb` class handles an exception. Take a look at this code before moving on to the next section.

## Class Review: `AssemblerEventHandler.java`

The `AssemblyEventHandler` class is an interface that defines the basic requirements to implement an assembler callback handler. This class is designed to provide a chance to customize an existing assembler implementation by adding code that responds to events in the assembly process. By default we have two such events that we can respond to. Let's take a look.

*Listing 10-4. `AssemblerEventHandler.java` Full Interface Review*

```

1 package net.middlemind.GenAsm.Assemblers;
2
3 import java.util.List;
4 import net.middlemind.GenAsm.JsonObjs.JsonObjIsSet;
5
6 public interface AssemblerEventHandler {
```

```
7     public void RunAssemblerPre(Assembler assembler, JsonObjectIsSet
jsonIsSet, String assemblySourceFile, List<String> assemblySource, String
outputDir, Object otherObj);
8     public void RunAssemblerPost(Assembler assembler);
9 }
```

A complete listing of the AssemblerEventHandler interface.

The first method signature we'll look at is the `RunAssemblerPre` method. This method takes a number of arguments and is called prior to the assembler's execution. We'll review the method's arguments now. The first is a reference to the assembler class, `assembler`. The second is a reference to the loaded instruction set data files, `jsonIsSet`, while the third argument is the path to the assembly source code file that's being processed, `assemblySourceFile`.

Next up, similar to the Assembler interfaces' `RunAssembler` method, we have the `assemblySource` argument, the desired output directory, and special argument for an unspecified object, `otherObj`, that can be cast out and used to customize your assembler implementation further. In the next section we'll look at this class in action.

## Demonstration: AssemblerEventHandler.java

The `AssemblerEventHandler` class is a bit tough to find a good example for. This is because we don't really use it in our implementation. The best example I could find was using the class as the bases for a Thumb-1 specific version of the interface.

*Listing 10-5. AssemblerEventHandler.java Demonstration*

File: `AssemblerEventHandlerThumb.java`  
Method: n/a  
Snippet:

```
01 package net.middlemind.GenAsm.Assemblers.Thumb;
02
03 import java.util.List;
04 import net.middlemind.GenAsm.AssemblersAssembler;
```

```

05 import net.middlemind.GenAsm.AssemblersAssemblerEventHandler;
06 import net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsFile;
07 import net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsValidLine;
08 import net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsValidLines;
09 import net.middlemind.GenAsm.Tokeners.TokenLine;
10
11 public interface AssemblerEventHandlerThumb extends
AssemblerEventHandler { }

```

A demonstration of the AssemblerEventHandler interface in use.

In the previous listing is an example of the AssemblerEventHandler interface is in use as part of the definition of a new, Thumb-1, specific interface, the AssemblerEventHandlerThumb interface. This means that the AssemblerEventHandlerThumb interface defines the required methods defined in the AssemblerEventHandler class as well as it's own required methods. Take a look at the interface and make sure you understand it before moving on. In the next section we'll look at some classes that are part of the Assembler package.

## Class Review: Symbols.java

The Symbols class is a simple class designed to hold the symbols defined in an assembly source code file. Symbols are defined by labels and by the @EQU directive. This class is concise, so I'll list it in its entirety here.

*Listing 10-6. Symbols.java Full Class Review*

```

1 package net.middlemind.GenAsm.Assemblers;
2
3 import java.util.Hashtable;
4 import java.util.Map;
5
6 public class Symbols {
7     public String obj_name = "Symbols";
8     public Map<String, Symbol> symbols = new Hashtable<>();
9 }

```

A complete listing of the Symbols class.

The class has only two public fields as members. The first is the `obj_name` field that is used to provide a proper name for the class in much the same way as the JSON objects we've seen during the JSON data file review. The second field is a little bit more interesting. The `symbols` field is a `Map` that allows you to store a `String, Symbol` associated pair. This is perfect for storing `Symbol` objects by name, just like they are used in an assembly source code file. In the next section we'll quickly take a look at this class in use.

## Demonstration: Symbols.java

To demonstrate the `Symbols` class, we'll look at a snippet of code from the `AssemblerThumb` class' `PopulateDirectiveArgAndAreaData` method. In this particular example we're looking at the code around the processing of a new symbol.

*Listing 10-7. Symbols.java Demonstration*

File: `AssemblerThumb.java`  
Method: `PopulateDirectiveArgAndAreaData`  
Snippet:

```
1 symbol.value = Utils.ParseNumberString(token.source);
2 symbols.symbols.put(lastLabel, symbol);
3 Logger.wrl("AssemblerThumb: PopulateDirectiveArgAndAreaData: Storing
symbol with label '" + lastLabel + "' for line number " +
lastLabelLine.lineNumAbs + " with value " + symbol.value);
```

An example of the `Symbols.java` class in use.

In the previously listed example, an instance of the `Symbol` class has its `value` field updated, line 1, with the results of a call to the `Utils` class' `ParseNumberString` method. On line 2 the symbol is then stored, by the name of the associated label, in an instance of the `Symbols` class named `symbols`. The example concludes with a logging call on line 3. We'll further this review by looking into the `Symbol` class next.

# Class Review: Symbol.java

The `Symbol` class is used to hold symbols defined with labels or with the `@EQU` directive. I should note that the concept of an associated value comes from the `@EQU` use case, otherwise we're more concerned with the line number of the symbol. The class is designed to handle a few different types of symbols. This is a pattern you'll see in many of the assembler's associated classes.

*Listing 10-8. Symbol.java Full Class Review*

```
01 package net.middlemind.GenAsm.Assemblers;
02
03 import net.middlemind.GenAsm.Tokeners.TokenLine;
04 import net.middlemind.GenAsm.Tokeners.Token;
05
06 public class Symbol {
07     public String obj_name = "Symbol";
08     public String name;
09     public TokenLine line;
10     public Token token;
11     public String addressHex;
12     public String addressBin;
13     public int addressInt;
14     public int lineNumActive;
15     public int lineNumAbs;
16     public boolean isEmptyLineLabel;
17     public boolean isLabel;
18     public boolean isStaticValue;
19     public Integer value;
20 }
```

A complete listing of the `Symbol` class.

The first field in the `Symbol` class is the `obj_name` field, it's used to hold the proper name of this class and also ensures this class displays its name when converted to JSON format. The `name` field is the way the symbol is stored and referenced. The next two fields, `line` and `token`, are references to the tokenized assembly source code line and the token in the

line representing this symbol. The next three fields, lines 11 – 13, hold the address, if any, of the symbol entry in different formats for convenience. The `lineNumActive` field is the line number this symbol is located on when only considering lines that are represented in the final binary output file.

The `lineNumAbs` field is a little different. This is the absolute line number that the symbol is located on when considering all lines in the assembly source code file. The next three Boolean fields are used to indicate the type of symbol the object instance represents. Lastly, the `value` field is used to hold the value associated with the symbol, if any. This only occurs in the case of an `@EQU` directive.

## Demonstration: Symbol.java

To demonstrate the `Symbol` class in use we'll look at a snippet of code from the `AssemblerThumb` class' `PopulateDirectiveArgAndAreaData` method. In this example the `@EQU` directive is being handled. If the symbols Map does not contain the current label, `lastLabel`, then we create a new symbol, lines 6 – 14, ultimately to be added to the list of symbols for the current assembly program.

*Listing 10-9. Symbol.java Demonstration*

File: `AssemblerThumb.java`

Method: `PopulateDirectiveArgAndAreaData`

Snippet:

```
01 } else if (token.source.equals(JsonObjIsDirectives.NAME_EQU)) {  
02     line.isEmpty = true;  
03     if (symbols.symbols.containsKey(lastLabel)) {  
04         throw new ExceptionRedefinitionOfLabel("Found symbol '" +  
lastLabel + "' redefined on line " + lastLabelLine.lineNumAbs + "  
originally defined on line " +  
(symbols.symbols.get(lastLabel)).lineNumAbs);  
05     }  
06     symbol = new Symbol();  
07     symbol.line = line;  
08     symbol.lineNumAbs = line.lineNumAbs;  
09     symbol.addressBin = line.addressBin;  
10     symbol.addressHex = line.addressHex;  
11     symbol.addressInt = line.addressInt;
```

```
12     symbol.name = lastLabel;
13     symbol.token = lastLabelToken;
14     symbol.isStaticValue = true;
15
16 }
```

An example of the Symbol.java class in use.

There are some finer points in the previously listed snippet that I'd like to discuss. The first one is related to line 2. Notice that the `isLineEmpty` flag is set to true for this line of assembly code. Given that we're in a block of code meant to handle the `@EQU` directive can you think of a reason why we're setting the `isLineEmpty` flag to true? Well, if the line is an `@EQU` directive, as indicated, it doesn't make it into the final binary representation of the program so the token line is marked as being empty in that regard.

Secondly, notice that on lines 7 and 12 we are storing information about the line and token that are associated with the symbol. Lastly, on lines 9 – 11, the address fields of the symbol are set with data from the token line the symbol is created from. In the next section we'll review the project's exception classes.

## Package Review: Exceptions

The Exceptions package contains only the `ExceptionBase` class. This class serves as the basis for all the GenAsm project's exceptions. We'll review this class next.

## Class Review: ExceptionBase.java

The `ExceptionBase` class is used as a foundation for all project exceptions. The class is simple and only serves as a basis for the purpose of future proofing the code and allowing there to be an entry point for centralized code changes to the project's exceptions.

*Listing 10-10. ExceptionBase.java Full Class Review*

```
1 package net.middlemind.GenAsm.Exceptions;  
2  
3 public class ExceptionBase extends Exception {  
4     public ExceptionBase(String errorMessage) {  
5         super(errorMessage);  
6     }  
7 }
```

A complete listing of the ExceptionBase class.

The ExceptionBase class is not an exception but an extension of the Exception class, line 4. Note that the constructor has one line of code that calls the super class' constructor on line 5. That's all there is to the class. We'll further our class review with a demonstration in the next section.

## Demonstration: ExceptionBase.java

The demonstration of the ExceptionBase class is another exception class that extends it.

*Listing 10-11. ExceptionBase.java Demonstration*

File: ExceptionInvalidArea.java

Method: n/a

Snippet:

```
1 package net.middlemind.GenAsm.Exceptions.Thumb;  
2  
3 import net.middlemind.GenAsm.Exceptions.ExceptionBase;  
4  
5 public class ExceptionInvalidArea extends ExceptionBase {  
6     public ExceptionInvalidArea(String errorMessage) {  
7         super(errorMessage);  
8     }  
9 }
```

An example of the ExceptionBase.java class in use.

The `ExceptionBase` class is used as the super class of all the project's exceptions. As you can see in the previously listed example, on line 5, the `ExceptionInvalidArea` class extends the `ExceptionBase` class. Notice that on line 7 the super class' constructor is called with the default argument, the `errorMessage` string. This wraps up our short review of the `ExceptionBase` class and the `Exceptions` package.

## Package Review: JsonObjs

The `JsonObjs` package contains all of the holder classes in the GenAsm project. Holder classes are those classes that are designed to hold data, specifically from a JSON data load, also known as a loader class. There are a number of holder classes that are part of the set of base classes. Up next, we'll begin the class review with the `JsonObj` interface.

## Interface Review: JsonObj.java

Java interfaces are a tool for defining the minimum requirements a class must implement to support the interface' functionality without actually defining the class. In this case the interface is designed to provide some basic functionality to all `JsonObj` based classes.

*Listing 10-12. JsonObj.java Full Interface Review*

```
01 package net.middlemind.GenAsm.JsonObjs;
02
03 import net.middlemind.GenAsm.Exceptions.Thumb.ExceptionJsonObjLink;
04
05 public interface JsonObj {
06     public String GetLoader();
07     public void SetLoader(String s);
08     public String GetName();
09     public void SetName(String s);
10     public String GetFileName();
11     public void SetFileName(String s);
12     public void Print();
13     public void Print(String prefix);
14     public void Link(JsonObj linkData) throws ExceptionJsonObjLink;
15 }
```

The example shown in the previous listing contains the minimum set of methods that must be implemented to support the `JsonObj` interface. In this case we have a few pairs of get and set methods followed by some general use methods `Print` and `Link`. From the set of methods defined we can see that a class that implements this interface has to define a loader, name, and file name fields in some fashion in order to support the required get and set methods.

Subsequently, the interface requires that a `JsonObj` based class implement printing and linking methods. The `Print` method supports providing a prefix value that is used before each line printed. The `Link` method is used to connect a string value to a class. This is used to connect objects that are referenced as strings in JSON data to the actual Java object that was loaded from the JSON data. In order to do this a `JsonObj` instance is provided to supply a list of objects to search through.

## Demonstration: `JsonObj.java`

The `JsonObj` interface is used mainly as the basis for other holder, `JsonObj`, classes. The following code snippet shows the `JsonObj` class in use.

*Listing 10-13. `JsonObj.java` Demonstration 1*

File: `JsonObjBase.java`  
Method: n/a  
Snippet:

```
1 package net.middlemind.GenAsm.JsonObjs;
2
3 import net.middlemind.GenAsm.Exceptions.Thumb.ExceptionJsonObjLink;
4 import net.middlemind.GenAsm.Logger;
5
6 public class JsonObjBase implements JsonObj { }
```

An example of the `JsonObj.java` interface in use.

Notice that the `JsonObjBase` class implements the `JsonObj` interface. This is the main use of the `JsonObj` interface in the project. Let's further the example a bit by viewing the implementation of a holder class.

*Listing 10-14. JsonObj.java Demonstration 2*

File: `JsonObjIsSets.java`

Method: n/a

Snippet:

```
1 package net.middlemind.GenAsm.JsonObjs;
2
3 import java.util.List;
4 import net.middlemind.GenAsm.Logger;
5
6 public class JsonObjIsSets extends JsonObjBase { }
```

An example of the `JsonObjBase.java` class in use and by extension the `JsonObj.java` interface.

The previously shown listing demonstrates the standard declaration of a new holder class. Note that this class extends the `JsonObjBase` class which implements the `JsonObj` interface.

## Class Review: `JsonObjBase.java`

The `JsonObjBase` class implements the `JsonObj` interface and is used as the basis for all JSON data holder classes.

*Listing 10-15. `JsonObjBase.java` Full Class Review*

```
01 package net.middlemind.GenAsm.JsonObjs;
02
03 import net.middlemind.GenAsm.Exceptions.Thumb.ExceptionJsonObjLink;
04 import net.middlemind.GenAsm.Logger;
05
06
07 public class JsonObjBase implements JsonObj {
```

```
08     public String name;
09     public String fileName;
10     public String loader;
11
12     @Override
13     public String GetLoader() {
14         return loader;
15     }
16
17     @Override
18     public void SetLoader(String s) {
19         loader = s;
20     }
21
22     @Override
23     public String GetName() {
24         return name;
25     }
26
27     @Override
28     public void SetName(String s) {
29         name = s;
30     }
31
32     @Override
33     public String GetFileName() {
34         return fileName;
35     }
36
37     @Override
38     public void SetFileName(String s) {
39         fileName = s;
40     }
41
42     @Override
43     public void Print() {
44         Print("");
45     }
46
47     @Override
48     public void Print(String prefix) {
49         Logger.wrl(prefix + "Name: " + name);
50         Logger.wrl(prefix + "FileName: " + fileName);
```

```
51     Logger.wrl(prefix + "Loader: " + loader);
52 }
53
54 @Override
55 public void Link(JsonObj linkData) throws ExceptionJsonObjLink {
56     Logger.wrl("JsonObjBase: Link: Do nothing...");
57 }
58 }
```

A complete listing of the `JsonObjBase` class.

Notice that because the `JsonObjBase` class implements the `JsonObj` interface it implements a number of methods defined in the interface. On lines 12 – 40 the get and set methods defined in the `JsonObj` interface are defined. Also notice that class fields have been added to support the get and set methods, lines 8 – 10. The remaining methods, `Print` and `Link`, are defined to fulfill the interface's requirements. Note that the first print method is a pass through to the second print method that takes a prefix argument.

The first method simply calls the overloaded method with an empty string as a prefix. These methods have some default code to print out the values of the class fields. Lastly the `Link` method is defined but marked as not functional with a special logging call, line 56. In the next section we'll take a look at the `JsonObjBase` class in use. The lack of detailed implementation of the interface methods by the `JsonObjBase` class is due to the fact that it is a general class meant to be extended. In other words there isn't a JSON data file associated with this class.

## Demonstration: `JsonObjBase.java`

The `JsonObjBase` class is the super class used by all JSON data holder classes in the project. To demonstrate this class in action we'll take a look at the `JsonObjBitRep` class definition, listed subsequently.

*Listing 10-16. `JsonObjBase.java` Demonstration*

File: `JsonObjBitRep.java`

Method: n/a

Snippet:

```
01 public class JsonObjBitRep extends JsonObjBase {  
02     public String obj_name = "JsonObjBitRep";  
03     public String bit_string;  
04     public int bit_int;  
05     public int bit_len;  
06  
07     @Override  
08     public void Print() {  
09         Print("");  
10    }  
11  
12     @Override  
13     public void Print(String prefix) {  
14         super.Print(prefix);  
15         Logger.wrl(prefix + "ObjName: " + obj_name);  
16         Logger.wrl(prefix + "BitString: " + bit_string);  
17         Logger.wrl(prefix + "BitInt: " + bit_int);  
18         Logger.wrl(prefix + "BitLen: " + bit_len);  
19     }  
20 }
```

An example of the `JsonObjBase` class in use as part of the definition of the `JsonObjBitRep` class definition.

In the previously listed example, the `JsonObjBitRep` class is defined as an extension of the `JsonObjBase` class. This means that the `JsonObjBitRep` class inherits any functionality defined by the `JsonObjBase` class. As you can see in the listing the `obj_name` field is defined along with a few new class fields that have been added. Using this implementation approach, we don't need to rework any of the get and set methods defined in the super class but, we do need to adjust the `Print` methods defined by the `JsonObjBase` class and redefine them to print information pertinent to the `JsonObjBitRep` class, lines 8 – 19.

## Class Review: JsonObjBitRep.java

The `JsonObjBitRep` class is an extension of the `JsonObjBase` class as we've just seen in the previous class review. I'll list the class definition again here for you to take a look at it.

*Listing 10-17. JsonObjBitRep.java Full Class Review*

```
01 public class JsonObjBitRep extends JsonObjBase {  
02     public String obj_name = "JsonObjBitRep";  
03     public String bit_string;  
04     public int bit_int;  
05     public int bit_len;  
06  
07     @Override  
08     public void Print() {  
09         Print("");  
10     }  
11  
12     @Override  
13     public void Print(String prefix) {  
14         super.Print(prefix);  
15         Logger.wrl(prefix + "ObjName: " + obj_name);  
16         Logger.wrl(prefix + "BitString: " + bit_string);  
17         Logger.wrl(prefix + "BitInt: " + bit_int);  
18         Logger.wrl(prefix + "BitLen: " + bit_len);  
19     }  
20 }
```

A full listing of the `JsonObjBitRep` class.

We took a look at this class as part of the demonstration section of the `JsonObjBase` class. The class' first field `bit_string` is used to hold the binary string represented by this class. The next field, `bit_int`, holds an integer representation of the binary number represented by the `bit_string` field. The last field, `bit_len`, is an integer that represents the length of the binary string being represented. In the next section we'll take a look at a demonstration of the class in use.

## Demonstration: JsonObjBitRep.java

We've encountered the `JsonObjBitRep` class before but as a JSON sub-object during our review of the project's data files.

*Listing 10-18. JsonObjBitRep.java Demonstration*

File: `JsonObjIsRegister.java`

Method: n/a

Snippet:

```
01 public JsonObjBitRep bit_rep;
```

File: `JsonObjIsOpCode.java`

Method: n/a

Snippet:

```
01 public JsonObjBitRep bit_rep;
```

File: `AssemblerThumb.java`

Method: `BuildBinOpCode`

Snippet:

```
01
02     if(entry.tokenOpCodeArgGroup.type_name.equals(JsonObjIsEntryTypes.NAME_REG
03           ISTER_LOW)) {
04         resTmp1 = entry.tokenOpCodeArgGroup.register.bit_rep.bit_string;
05
06     } else
07     if(entry.tokenOpCodeArgGroup.type_name.equals(JsonObjIsEntryTypes.NAME_REG
08           ISTER_HI)) {
09         resTmp1 = entry.tokenOpCodeArgGroup.register.bit_rep.bit_string;
10
11    } else
12    if(entry.tokenOpCodeArgGroup.type_name.equals(JsonObjIsEntryTypes.NAME_REG
13           ISTER_PC)) {
14        resTmp1 = entry.tokenOpCodeArgGroup.register.bit_rep.bit_string;
15
16    } else
17    if(entry.tokenOpCodeArgGroup.type_name.equals(JsonObjIsEntryTypes.NAME_REG
18           ISTER_SP)) {
19        resTmp1 = entry.tokenOpCodeArgGroup.register.bit_rep.bit_string;
20
21    }
```

```

13 } else
if(entry.tokenOpCodeArgGroup.type_name.equals(JsonObjIsEntryTypes.NAME_REG
ISTER_LR)) {
14     resTmp1 = entry.tokenOpCodeArgGroup.register.bit_rep.bit_string;
15
16 }

```

An example of the `JsonObjBitRep` class being used as a field in other JSON data holder classes followed by a specific use case.

In the previously listed example, the `JsonObjBitRep` class is used to define the field, `bit_rep`, of another JSON data holder class. Using this approach, the JSON data loading library, `Gson`, can automatically convert JSON objects to their associated Java class representation. In this way the more general, base, classes can be used in the definition of other more complex JSON data holder classes. The last example in the previous listing demonstrates using the class in the `AssemblerThumb` class' `BuildBinOpCode` method. In this code snippet a selection of high registers are converted to their associated binary representation, lines 1 – 16.

## Class Review: `JsonObjBitSeries.java`

The `JsonObjBitSeries` class is another sub-object holder class. We've encountered this sub-object during our review of the Thumb-1 instruction set's data files.

*Listing 10-19. `JsonObjBitSeries.java` Full Class Review*

```

01 package net.middlemind.GenAsm.JsonObjs;
02
03 import net.middlemind.GenAsm.Logger;
04
05 public class JsonObjBitSeries extends JsonObjBase {
06     public String obj_name = "JsonObjBitSeries";
07     public int bit_start;
08     public int bit_stop;
09     public int bit_len;
10
11     @Override

```

```

12     public void Print() {
13         Print("");
14     }
15
16     @Override
17     public void Print(String prefix) {
18         super.Print(prefix);
19         Logger.wrl(prefix + "ObjName: " + obj_name);
20         Logger.wrl(prefix + "BitStart: " + bit_start);
21         Logger.wrl(prefix + "BitStop: " + bit_stop);
22         Logger.wrl(prefix + "BitLen: " + bit_len);
23     }
24
25     public void PrintShort() {
26         PrintShort("");
27     }
28
29     public void PrintShort(String prefix) {
30         Logger.wrl(prefix + "BitStart: " + bit_start);
31         Logger.wrl(prefix + "BitStop: " + bit_stop);
32         Logger.wrl(prefix + "BitLen: " + bit_len);
33     }
34 }
```

A full listing of the `JsonObjBitSeries` class.

The `JsonObjBitSeries` class defines the `obj_name` field, providing it with a default value, on line 6. The next three fields, `bit_start`, `bit_stop`, and `bit_len` are used to map out the binary series. Notice that the class redefines the `Print` methods, customizing them to print bit series specific information. The class also defines an abridged version of the print methods with the addition of the `PrintShort` methods. This shortened version of the print method works in much the same way as the print methods we've seen earlier. Make sure to look over the class before moving on to the next section where we demonstrate the class in use.

## Demonstration: JsonObjBitSeries.java

The best way to demonstrate this class is to show it in use as a class field for some of the other JSON data holder classes. I should detail the types of JSON objects and their nomenclature a bit in case it hasn't become apparent.

*Listing 10-20. JSON Objects to Java Classes*

JSON Object	JSON Object Type	Java Class	Java Class Type
{ "bit_start": 0, "bit_stop": 31, "bit_len": 32 }	Sub-Object	JsonObjBitSeries	Assembler - JSON Data Holder Class
{ "obj_name": "is_op_code_arg", "arg_index": 1, "bit_index": 0, "is_entry_types": [ ... ], "is_arg_type": "Word8", "bit_series": { ... }, "num_range": { ... }, "bit_shift": { ... } }	Sub-Object	JsonObjIsOpCodeArg	Assembler - JSON Data Holder Class
{ "obj_name": "line_hex_reps", "line_hex_reps": [ {"addressHex": "080 000F4", "valueHex": "2A49"} ] }	Object	JsonObjLineHexReps	Unit Test - JSON Data Holder Class

```
,
```

```
...
```

```
]
```

```
}
```

A table that shows the names of different JSON object types and the Java class type that is associated with it.

The `JsonObjBitSeries` class can be found in use in the following JSON data holder classes.

*Listing 10-21. JsonObjBitSeries.java Demonstration*

File: `JsonObjIsOpCodeArg.java`

Method: n/a

Snippet:

```
01 public JsonObjBitSeries bit_series;
```

File: `JsonObjIsOpCodes.java`

Method: n/a

Snippet:

```
01 public JsonObjBitSeries bit_series;
```

File: `JsonObjIsOpCode.java`

Method: n/a

Snippet:

```
01 public JsonObjBitSeries bit_series;
```

File: `JsonObjIsDirectiveArg.java`

Method: n/a

Snippet:

```
01 public JsonObjBitSeries bit_series;
```

File: `JsonObjIsOpCodeArgSorter.java`

Method: compare

Snippet:

```
01 public int compare(JsonObjIsOpCodeArg a, JsonObjIsOpCodeArg b) {  
02     if(sortType == JsonObjIsOpCodeArgSorterType.BIT_SERIES_ASC) {  
03         return (a.bit_series.bit_start - b.bit_series.bit_start);  
04     }  
05 }
```

```

04     } else if(sortType == JsonObjIsOpCodeArgSorterType.BIT_SERIES_DSC) {
05         return (b.bit_series.bit_start - a.bit_series.bit_start);
06     } else if(sortType == JsonObjIsOpCodeArgSorterType.BIT_INDEX_ASC) {
07         return (a.bit_index - b.bit_index);
08     } else if(sortType == JsonObjIsOpCodeArgSorterType.BIT_INDEX_DSC) {
09         return (b.bit_index - a.bit_index);
10    } else if(sortType == JsonObjIsOpCodeArgSorterType.ARG_INDEX_DSC) {
11        return (b.arg_index - a.arg_index);
12    } else {
13        return (a.arg_index - b.arg_index);
14    }
15 }

```

An example of the `JsonObjBitSeries` class being used as a field in other JSON data holder classes followed by a specific use case.

The examples shown in the previous listing demonstrate the `JsonObjBitSeries` class in use as part of a few other JSON data holder classes. The last example from the previous listing shows the `JsonObjIsOpCodeArgSorter` class' `compare` method. Note the code on lines 2 – 5 that checks to see which argument, `a` or `b`, occurs first when ordered ascending or descending. If you recall from our review of the JSON data files a bit series object helps order the opcode arguments in different ways. In the next section, we'll review another sub-object class that we've seen before during the review of the JSON data files. The `JsonObjBitShift` class.

## Class Review: `JsonObjBitShift.java`

The `JsonObjBitShift` class is a JSON data holder class that is designed to hold bit shift JSON sub-objects. We've seen this JSON object before when we reviewed the project's JSON data files.

*Listing 10-22. `JsonObjBitShift.java` Full Class Review*

```

01 package net.middlemind.GenAsm.JsonObjs;
02
03 import net.middlemind.GenAsm.Logger;
04

```

```

05 public class JsonObjBitShift extends JsonObjBase {
06     public String obj_name = "JsonObjBitShift";
07     public String shift_dir;
08     public int shift_amount;
09
10    @Override
11    public void Print() {
12        Print("");
13    }
14
15    @Override
16    public void Print(String prefix) {
17        super.Print(prefix);
18        Logger.wrl(prefix + "ObjName: " + obj_name);
19        Logger.wrl(prefix + "ShiftDir: " + shift_dir);
20        Logger.wrl(prefix + "ShiftAmount: " + shift_amount);
21    }
22 }
```

A full listing of the `JsonObjBitShift` class.

In the same way we've seen before, the `JsonObjBitShift` object defines the `obj_name` field along with the `shift_dir` and `shift_amount` fields needed to define the bit shift. In case it hasn't jumped out at you the Java data holder classes have the same fields defined as the JSON object. Also, as we'd expect, the `Print` methods are overridden to redefine them so we can include the class fields in the print operation.

## Demonstration: `JsonObjBitShift.java`

To demonstrate the `JsonObjBitShift` class in action we've shown a few examples in the following listing.

*Listing 10-23. `JsonObjBitShift.java` Demonstration*

File: `JsonObjIsOpCodeArg.java`  
 Method: n/a  
 Snippet:

```

1 public JsonObjBitShift bit_shift;
```

File: JsonObjIsDirectiveArg.java

Method: n/a

Snippet:

```
1 public JsonObjBitShift bit_shift;
```

File: AssemblerThumb.java

Method: BuildBinDirective

Snippet:

```
1 if(!Utils.IsEmpty(entry.bit_shift.shift_dir) &&
entry.bit_shift.shift_dir.equals(NUMBER_SHIFT_NAME_LEFT)) {
2     resTmp = Utils.ShiftBinStr(resTmp, entry.bit_shift.shift_amount,
false, true);
3 } else if(!Utils.IsEmpty(entry.bit_shift.shift_dir) &&
entry.bit_shift.shift_dir.equals(NUMBER_SHIFT_NAME_RIGHT)) {
4     resTmp = Utils.ShiftBinStr(resTmp, entry.bit_shift.shift_amount,
true, true);
5 } else {
6     throw new ExceptionNumberInvalidShift("Invalid number shift found for
source '" + token.source + "' with line number " + token.lineNumAbs);
7 }
```

An example of the `JsonObjBitShift` class being used as a field in other JSON data holder classes followed by a specific use case.

In the previous listing we can see that the class is used as a field in two other JSON data holder classes, the `JsonObjIsOpCodeArg` and `JsonObjIsDirectiveArg` classes. In the final example of the listing, we can see the class in use to drive a bit shift operation on lines 2 and 4. Notice that if the shift operation doesn't match up properly, or is unsupported, an exception is thrown on line 6.

## Class Review: `JsonObjIsSets.java`

The `JsonObjIsSets` class is an important JSON data holder class as it is used to hold all of the loaded instruction set definitions, including all the associated data files. Let's take a look at the class' source code, shown in the next listing.

### *Listing 10-24. JsonObjIsSets.java Full Class Review*

```
01 package net.middlemind.GenAsm.JsonObjs;
02
03 import java.util.List;
04 import net.middlemind.GenAsm.Logger;
05
06 public class JsonObjIsSets extends JsonObjBase {
07     public String obj_name = "JsonObjIsSets";
08     public List<JsonObjIsSet> is_sets;
09
10    @Override
11    public void Print() {
12        Print("");
13    }
14
15    @Override
16    public void Print(String prefix) {
17        super.Print(prefix);
18        Logger.wrl(prefix + "ObjectName: " + obj_name);
19
20        Logger.wrl(prefix + "IsSets:");
21        for(JsonObjIsSet entry : is_sets) {
22            Logger.wrl("");
23            entry.Print(prefix + "\t");
24        }
25    }
26 }
```

A full listing of the JsonObjIsSets class.

This version of the JSON data holder class is used to hold a list of `JsonObjIsSet` sub-objects. Note the `obj_name` field and default value. This value should be overwritten if the JSON object loaded has the `obj_name` attribute defined. The second class field, `is_sets`, is used to hold the list of sub-objects. Lastly, the class overwrites the default print methods to customize them for this particular class' fields. In particular the print method supports iterating over the `is_sets` list and printing information about each entry.

## Demonstration: JsonObjIsSets.java

The `JsonObjIsSets` class is an important data holder class because it holds references to all supported instructions sets and their associated data files. Let's take a look at the class in action.

*Listing 10-25. JsonObjIsSets.java Demonstration*

File: GenAsm.java

Method: n/a

Snippet:

```
1 public static JsonObjIsSets ASM_SETS = null;
```

File: LoaderIsSets.java

Method: ParseJson

Snippet:

```
1 JsonObjIsSets jsonObj =
(JsonObjIsSets)Class.forName(targetClass).getConstructor().newInstance();
2 jsonObj = gson.fromJson(json, jsonObj.getClass());
3 jsonObj.name = targetClass;
4 jsonObj.fileName = fileName;
5 jsonObj.loader = getClass().getName();
```

A few examples of the `JsonObjIsSets` class in use.

In the first example a line of code from the `GenAsm` class is shown. The `ASM_SETS` is a static field that holds a reference to the loaded instruction sets. This field is used in the early steps of the assembly process. The second example shown is from the `LoaderIsSets` class' `ParseJson` method. This little snippet of code is important because it demonstrates how the data driven class name is cast to the expected `JsonObjIsSets` class type.

This shows us that the code is somewhat extensible, but you have to explicitly support for the classes you use in your JSON data files. Also notice that the class fields, `name`, `fileName`, and `loader`, are populated as part of the data load process. In the next section we'll take a look at the `JsonObjIsSet` sub-object.

## Class Review: JsonObjIsSet.java

The next class for us to look at is the `JsonObjIsSet` class. This class is the sub-object associated with the `JsonObjIsSets` class we just finished reviewing. Recall that the `JsonObjIsSets` instances are stored in the `is_sets` field of the `JsonObjIsSet` class.

*Listing 10-26. JsonObjIsSet.java Full Class Review*

```
01 package net.middlemind.GenAsm.JsonObjs;
02
03 import java.util.List;
04 import net.middlemind.GenAsm.Logger;
05
06 public class JsonObjIsSet extends JsonObjBase {
07     public String obj_name = "JsonObjIsSet";
08     public String set_name;
09     public List<JsonObjIsFile> is_files;
10
11     @Override
12     public void Print() {
13         Print("");
14     }
15
16     @Override
17     public void Print(String prefix) {
18         super.Print(prefix);
19         Logger.wrl(prefix + "ObjectName: " + obj_name);
20         Logger.wrl(prefix + "SetName: " + set_name);
21
22         Logger.wrl(prefix + "IsFiles:");
23         for(JsonObjIsFile entry : is_files) {
24             Logger.wrl("");
25             entry.Print(prefix + "\t");
26         }
27     }
28 }
```

A full listing of the `JsonObjIsSet` class.

Notice this class has an `obj_name` field in much the same way we've seen before with JSON data holder classes. The next field, `set_name`, is used indicate which instruction set this object instance belongs to. The last field, line 9, is the `is_files` field. This field is used to keep track of all the data files associated with a given instruction set. This class overrides the default print methods to provide custom printing for this class.

## Demonstration: JsonObjIsSet.java

To demonstrate the `JsonObjIsSet` class we'll take a look at a few class methods that use it as well as some key snippets of code. Let's take a look.

*Listing 10-27. JsonObjIsSet.java Demonstration*

File: Assembler.java

Method: RunAssembler

Snippet:

```
01 public void RunAssembler(JsonObjIsSet jsonIsSet, String
assemblySourceFile, List<String> assemblySource, String outputDir, Object
otherObj, AssemblerEventHandler asmEventHandler, boolean verbose, boolean
quellOutput) throws Exception;
```

File: AssemblerEventHandler.java

Method: RunAssemblerPre

Snippet:

```
01 public void RunAssemblerPre(Assembler assembler, JsonObjIsSet
jsonIsSet, String assemblySourceFile, List<String> assemblySource, String
outputDir, Object otherObj);
```

File: AssemblerThumb.java

Method: n/a

Snippet:

```
01 public JsonObjIsSet isaDataSet;
```

File: LoaderIsSets.java

Method: ParseJson

Snippet:

```
01 JsonObjIsSets jsonObj =
(JsonObjIsSets)Class.forName(targetClass).getConstructor().newInstance();
02 jsonObj = gson.fromJson(json, jsonObj.getClass());
03 jsonObj.name = targetClass;
```

```

04 jsonObj.fileName = fileName;
05 jsonObj.loader = getClass().getName();
06
07 for(JsonObjIsSet entry : jsonObj.is_sets) {
08     entry.name = entry.getClass().getName();
09     entry.fileName = fileName;
10     entry.loader = getClass().getName();
11
12     for(JsonObjIsFile fentry : entry.is_files) {
13         fentry.name = fentry.getClass().getName();
14         fentry.fileName = fileName;
15         fentry.loader = getClass().getName();
16     }
17 }

```

File: GenAsm.java

Method: main

Snippet:

```

01 for(JsonObjIsSet entry : ASM_SETS.is_sets) {
02     if(entry.set_name.equals(ASM_TARGET_SET)) {
03         Logger.wrl("GenAsm: Main: Found instruction set entry " +
ASM_TARGET_SET);
04         ASM_SET = entry;
05         break;
06     }
07 }

```

A few examples of the `JsonObjIsSet` class in use.

The first two examples shown in the previous listing should be familiar to you as we've looked at these classes in this chapter. The important thing to note here is that the `JsonObjIsSet` class is being used as a definition of the instruction set and is passed into the `RunAssemblerPre` and `RunAssembler` methods.

The third example, shown previously, is from an instruction set specific class the `AssemblerThumb` class. In the example line of code, we can see a class field, `isaDataSet`, that is used to reference an instance of the `JsonObjIsSet` class. The third example shown is from the `LoaderIsSets` class' `ParseJson` method.

In the snippet of code shown for this example we can see how the `JsonObjIsSets` class is initialized, and its fields are set, on lines 1 – 5. On lines 7 – 17 the loader iterates over each member object, instances of the `JsonObjIsSet` class, and its member object, instances of the `JsonObjIsFile` class, setting base field values as it goes. The last example shown is from the `GenAsm` class' main method. This example shows a snippet of code used to load up a target instruction set based on the value of a program argument.

On line 2 of the example snippet if we've found a `JsonObjIsSet` entry that has the same `set_name` as the program argument, `ASM_TARGET_SET`, the `ASM_SET` field is updated, and the loop exits. This little bit of code is used to locate and load up the specified instruction set for use in the assembly process. Notice that it tries to find a match from the list of loaded file sets.

## Class Review: `JsonObjIsFile.java`

The next class for us to review is the member object of the `JsonObjIsSet` class we just reviewed. It contains a list of files that are part of an instruction set definition. The class that holds the instruction set file data is as follows.

*Listing 10-28. `JsonObjIsFile.java` Full Class Review*

```
01 package net.middlemind.GenAsm.JsonObjs;
02
03 import net.middlemind.GenAsm.Logger;
04
05 public class JsonObjIsFile extends JsonObjBase {
06     public String obj_name;
07     public String path;
08     public String loader_class;
09     public String target_class;
10     public String category;
11
12     @Override
13     public void Print() {
14         Print("");
15     }
16 }
```

```

17     @Override
18     public void Print(String prefix) {
19         super.Print(prefix);
20         Logger.wrl(prefix + "ObjectName: " + obj_name);
21         Logger.wrl(prefix + "Path: " + path);
22         Logger.wrl(prefix + "LoaderClass: " + loader_class);
23         Logger.wrl(prefix + "TargetClass: " + target_class);
24         Logger.wrl(prefix + "ObjectName: " + category);
25     }
26 }
```

A full listing of the `JsonObjIsFile` class.

First, note the `obj_name` field. This time there is no default value set but, in any case, those values will be overwritten with whatever data is found in the JSON object's `obj_name` attribute. The second field `path`, found on line 7, is used to hold a full path to the target file. The next field, `loader_class`, is used to define the full package and class name of the loader class that should be used to load this file.

Similarly, the `target_class` field, line 9, is the full package and class name of the holder class designed to hold the data contained in the specified file. The ARM Thumb-1 instruction set, as implemented in the GenAsm assembler, uses JSON files to store its data, but this is not a requirement. As long as your loader class can process the data file and populate a holder class, things should still work fine.

The last field in this class, `category`, is used to provide a little bit of descriptive information with regard to the target data file specified by the class. As we've seen before in other holder classes the print methods have been redefined to support customizing the print methods. In the next section we'll look at an example of this class in action.

## Demonstration: `JsonObjIsFile.java`

To demonstrate the `JsonObjIsFile` class we'll take a look at a few places in the project's code base that the class is used.

*Listing 10-29. JsonObjIsFile.java Demonstration*

File: AssemblerThumb.java

Method: LoadAndParseJsonObjData

Snippet:

```
1 for(JsonObjIsFile entry : isaDataSet.is_files) { }
```

File: AssemblerEventHandlerThumb.java

Method:

Snippet:

```
1 public void LoadAndParseJsonObjDataLoopPre(int step, Assembler
assembler, JsonObjIsFile entry);
2 public void LoadAndParseJsonObjDataLoopPost(int step, Assembler
assembler, JsonObjIsFile entry);
```

File: LoaderIsSets.java

Method: ParseJson

Snippet:

```
1 for(JsonObjIsFile fentry : entry.is_files) {
2     fentry.name = fentry.getClass().getName();
3     fentry.fileName = fileName;
4     fentry.loader = getClass().getName();
5 }
```

A few examples of the JsonObjIsFile class in use.

The first example shown in the previous listing is from the AssemblerThumb class' LoadAndParseJsonObjData method. In line 1 of the example, we iterate over the entries in the is\_files list for a given instruction set. In the second example shown, from the AssemblerEventHandlerThumb class, there are two methods defined that take a JsonObjIsFile object as an argument, line 1 and 2. Lastly, the loader class for these holder objects, LoaderIsSets, is the third example shown. In this example, the files are iterated over and for each JsonObjIsFile entry found the name, fileName, and loader class fields are set.

## Class Review: JsonObjLineHexReps.java

The `JsonObjLineHexReps` class is a little different than the classes we've reviewed thus far in this chapter. Up till now the classes we've reviewed have been in some way related to the data files that are used to define an instruction set. In this case, the `JsonObjLineHexReps`, class is used to hold the binary encodings of a given line number for an assembly program. The class is used as part of the project's unit tests not as part of an instruction set definition.

*Listing 10-30. JsonObjLineHexReps.java Full Class Review*

```
01 package net.middlemind.GenAsm.JsonObjs;
02
03 import java.util.List;
04 import net.middlemind.GenAsm.Logger;
05
06 public class JsonObjLineHexReps extends JsonObjBase {
07     public String obj_name = "JsonObjLineHexReps";
08     public List<JsonObjLineHexRep> line_hex_reps;
09
10    @Override
11    public void Print() {
12        Print("");
13    }
14
15    @Override
16    public void Print(String prefix) {
17        super.Print(prefix);
18        Logger.wrl(prefix + "ObjectName: " + obj_name);
19
20        Logger.wrl(prefix + "LineHexRep:");
21        for(JsonObjLineHexRep entry : line_hex_reps) {
22            Logger.wrl("");
23            entry.Print(prefix + "\t");
24        }
25    }
26 }
```

A full listing of the `JsonObjLineHexReps` class.

This class follows the pattern of a JSON data holder class that has sub-objects. Let's take a look. The first thing that should stand out is that we're sticking with the `obj_name` field convention. It's the first field defined on line 7 and defaulted to the name of the class. This field will be overwritten by a JSON object's `obj_name` attribute during data load. The class field `line_hex_reps`, line 8, is a list of `JsonObjLineHexRep` object instances used to represent the assembled, hexadecimal, value of each line of an assembly program. The class' print methods have been overwritten to allow for customization of the print process for this specific class.

## Demonstration: `JsonObjLineHexReps.java`

To demonstrate the class, we'll look at a few examples in the subsequent listing. Take a close look at each example before we review them.

*Listing 10-31. `JsonObjLineHexReps.java` Demonstration*

File: `Utils.java`

Method: `CheckAssemblerTestProgramAgainstAnswers`

Snippet:

```
01 for(String key : hashMap.keySet()) {  
02     String val = hashMap.get(key);  
03     key = key.toUpperCase();  
04     val = val.toUpperCase();  
05     boolean found = false;  
06     for(JsonObjLineHexRep hexLine : hexDataLines.line_hex_reps) {  
07         String aKey = hexLine.addressHex.toUpperCase();  
08         String aVal = hexLine.valueHex.toUpperCase();  
09         if(aKey.equals(key) && aVal.equals(val)) {  
10             found = true;  
11             break;  
12         }  
13     }  
14     ...  
15 }
```

File: `TestProgramA.java`

Method: `test1`

Snippet:

```
01 json = FileLoader.LoadStr(jsonAnswersDataFile);
```

```

02 jsonObj = ldr.ParseJson(json, answersTargetClass, jsonAnswersDataFile);
03 jsonName = jsonObj.GetName();
04 JsonObjectLineHexReps hexDataLines = (JsonObjectLineHexReps)jsonObj;
05 Logger.LOGGING_ON = true;
06 Logger.wrl(testName + ": Found " + hexDataLines.line_hex_reps.size() +
" test program answers in the loaded JSON data file, '" + jsonName +
".'");
07 Logger.wrl(testName + ": Found " + linkerThumb.hexMapLe.size() + " test
program lines entries in the resulting linked data, '" +
assemblySourceFile + "'.");
08 res = Utils.CheckAssemblerTestProgramAgainstAnswers(hexDataLines,
linkerThumb.hexMapLe);
09 Logger.wrl("CheckResults: " + res);

```

File: LoaderLineHexReps.java

Method: ParseJson

Snippet:

```

01 JsonObjectLineHexReps jsonObj =
(JsonObjectLineHexReps)Class.forName(targetClass).getConstructor().newInstance();
02 jsonObj = gson.fromJson(json, jsonObj.getClass());
03 jsonObj.name = targetClass;
04 jsonObj.fileName = fileName;
05 jsonObj.loader = getClass().getName();

```

A few examples of the JsonObjectLineHexReps class in use.

The first example, in the previous listing, is a snippet of code from the `Utils` class' `CheckAssemblerTestProgramAgainstAnswers` method. This method provides a centralized way to compare the results of assembling a program against a known example. In this case, with regard to project unit tests, we use the results from a VASM version of the program. On line 1 of the example snippet, we iterate over the `Map` extracting the `key`, `address`, and `value`, `hex number`, of each entry, lines 2 – 4.

Next, on line 5, we set a local variable to track if we've found a match or not, no need to keep looking if we've found one. On lines 6 – 13 we iterate over the contents of the `line_hex_reps` field of a `JsonObjectLineHexReps` object instance and search for a matching line number, value pair. If we've found a match we break out of the loop, lines 9 – 12.

The second example is from the `TestProgramA` class' `test1` method. This is a unit test method and as such we'll be performing a check to see if our test program assembled correctly. On lines 1 – 4 the comparison data is loaded. You can see that on line 4 the data holder class is cast down from the super class and we now have the data we need to perform the validation.

On line 8 the comparison is made with the newly loaded data and the little-endian assembly results, `linkerThumb.hexMapLe`, with a call to the `Utils` method that handles the validation. The last example shown is from the loading class associated with this data holder class, `LoaderLineHexReps`. The snippet of code shown is from the class' `ParseJson` method. It demonstrates initializing the `JsonObjLineHexReps` class, lines 1 – 2, and setting the value of its fields, lines 3 – 5.

In the next section we'll take a look at the sub-object associated with this class, the `JsonObjLineHexRep` class. This follows the naming convention where the parent class is plural, and the associated sub-object class is singular.

## Class Review: `JsonObjLineHexRep.java`

The `JsonObjLineHexRep` class is the sub-object class associated with the `JsonObjLineHexReps` class we just reviewed. This class is used to hold a hexadecimal line number and value association and is used as part of the project's unit tests.

*Listing 10-32. `JsonObjLineHexRep.java` Full Class Review*

```
01 package net.middlemind.GenAsm.JsonObjs;
02
03 import net.middlemind.GenAsm.Logger;
04
05 public class JsonObjLineHexRep extends JsonObjBase {
06     public String obj_name = "JsonObjLineHexRep";
07     public String addressHex;
08     public String valueHex;
09
10    @Override
11    public void Print() {
```

```

12         Print("");
13     }
14
15     @Override
16     public void Print(String prefix) {
17         super.Print(prefix);
18         Logger.wrl(prefix + "ObjName: " + obj_name);
19         Logger.wrl(prefix + "AddressHex: " + addressHex);
20         Logger.wrl(prefix + "ValueHex: " + valueHex);
21     }
22 }
```

A complete listing of the `JsonObjLineHexRep` class.

The `JsonObjLineHexRep` class, shown in the previous listing, is very similar to the data holder classes we've seen before. As such we won't go into too much detail while reviewing it. I'll simply point out the class fields `addressHex` and `valueHex` align with the purpose of the class as a data source for comparing the assembler's output for a given line of assembly source code.

## Demonstration: `JsonObjLineHexRep.java`

Because the `JsonObjLineHexRep` class is so concise, and its use limited to the project's unit tests. We won't have anything new to add to the demonstration we've already reviewed for its parent class, `JsonObjLineHexReps`. I will, however, list the most important use of the class here.

*Listing 10-33. `JsonObjLineHexRep.java` Demonstration*

```

1 for(JsonObjLineHexRep hexLine : hexDataLines.line_hex_reps) {
2     String aKey = hexLine.addressHex.toUpperCase();
3     String aVal = hexLine.valueHex.toUpperCase();
4     if(aKey.equals(key) && aVal.equals(val)) {
5         found = true;
6         break;
7     }
8 }
```

An example of the JsonObjLineHexRep class in use.

In the previously shown snippet the loop on line 1 iterates over a list of JsonObjLineHexRep objects. For each entry the key, address, and value, hex value, are extracted on lines 2 – 3. The comparison on line 4 determines if the addresses and values match and if so, exits the loop. We're almost done with the JSON holder class review so hang in there. There are only two more classes for us to look at.

## Class Review: JsonObjNumRange.java

The next JSON data holder class for us to review is the JsonObjNumRange class. This class is used to define a valid numeric range used to set constraints on certain opcode arguments. We've encountered this sub-object during our review of the ARM Thumb-1 instruction set. Let's take a look.

*Listing 10-34. JsonObjNumRange.java Full Class Review*

```
01 package net.middlemind.GenAsm.JsonObjs;
02
03 import net.middlemind.GenAsm.Logger;
04
05 public class JsonObjNumRange extends JsonObjBase {
06     public String obj_name = "JsonObjNumRange";
07     public int min_value;
08     public int max_value;
09     public int bit_len;
10     public boolean twos_complement;
11     public boolean ones_complement;
12     public boolean bcd_encoding;
13     public boolean handle_prefetch;
14     public boolean use_halfword_offset;
15     public String alignment;
16
17     @Override
18     public void Print() {
19         Print("");
20     }
21 }
```

```

22     @Override
23     public void Print(String prefix) {
24         super.Print(prefix);
25         Logger.wrl(prefix + "ObjName: " + obj_name);
26         Logger.wrl(prefix + "MinValue: " + min_value);
27         Logger.wrl(prefix + "MaxValue: " + max_value);
28         Logger.wrl(prefix + "BitLen: " + bit_len);
29         Logger.wrl(prefix + "TwosComplement: " + twos_complement);
30         Logger.wrl(prefix + "OnesComplement: " + ones_complement);
31         Logger.wrl(prefix + "BcdEncoding: " + bcd_encoding);
32         Logger.wrl(prefix + "HandlePrefetch: " + handle_prefetch);
33         Logger.wrl(prefix + "UseHalfWordOffset: " + use_halfword_offset);
34         Logger.wrl(prefix + "Alignment: " + alignment);
35     }
36 }
```

A complete listing of the `JsonObjNumRange` class.

The `JsonObjNumRange` class is similar to the JSON data holder classes we've seen previously. Due to the redundancy in the class structure, we'll focus on the aspects of the class that are different from the other classes we've reviewed, namely, the class' fields. The first entry is the `obj_name` field and is used to hold a proper name for the class.

The next two fields, `min_value` and `max_value`, describe the minimum and maximum value of the number range. The `bit_length` is used to describe the number of bits used to represent the number. The next five fields are used to give the assembler information about the numeric value including Booleans to indicate if a ones complement or twos complement needs to be applied.

The remaining three Booleans aren't really used by the assembler, but we kept them around because they might come in handy. The `bcd_encoding` field is a Boolean flag used to indicate if the number needs to be converted to a binary coded decimal value. A binary coded decimal number is a system of writing numerals that assigns a four-digit binary code to each digit 0 through 9 in a decimal number.

It is a way to convert decimal numbers into a binary value but is not the same as a simple binary representation of the same integer value. The next field, `handle_prefetch`, is a Boolean flag is used to indicate if the prefetch needs to be taken into account and lastly the `use_halfword_offset` field can be used to indicate a necessary binary offset.

Lastly the `alignment` field is a string value that indicates the desired binary alignment such as WORD or HALFWORD. If you think about the ARM Thumb-1 instruction set every entry takes 2 bytes, each line in an assembly source code file is WORD, 2 bytes, aligned, in other words it is located at an address divisible by two. On the other hand HALFWORD alignment means the address occurs at an odd position in the assembly source code file.

Please note that each line in an assembly source code file can be thought of as occurring at an address with a source cod file starting at address 0 and counting 2 bytes per valid line of code. The remainder of the class should be familiar to you. Take a moment to look it over before moving on to the next section.

## Demonstration: JsonObjNumRange.java

The `JsonObjNumRange` class is used in a few places throughout the project. It mainly comes into play when dealing with opcode or directive arguments. Let's take a look at some examples.

*Listing 10-35. JsonObjNumRange.java Demonstration*

File: AssemblerThumb.java

Method: LoadAndParseJsonObjData

Snippet:

```
1 lineNumRange = new JsonObjNumRange();
2 lineNumRange.alignment = "WORD";
3 lineNumRange.bcd_encoding = false;
4 lineNumRange.bit_len = lineBitSeries.bit_len;
5 lineNumRange.obj_name = "JsonObjNumRange";
6 lineNumRange.min_value = 0;
7 lineNumRange.max_value = 65536;
8 lineNumRange.ones_complement = false;
9 lineNumRange.twos_complement = false;
```

File: JsonObjIsOpCodeArg.java  
Method: n/a  
Snippet:  
1 public JsonObjNumRange num\_range;

File: JsonObjIsDirectiveArg.java  
Method: n/a  
Snippet:  
1 public JsonObjNumRange num\_range;

A few examples of the JsonObjNumRange class in use.

The first example shown in the previous listing is from the AssemblerThumb class' LoadAndParseJsonObjData method. In this case the JsonObjNumRange class field, lineNumberRange, is being initialized. This field is used to check the numeric values of arguments for certain standard directives and to provide a sanity check to make sure numeric arguments are reasonable.

The snippet of code associated with the first example, lines 1 – 9 in the previous listing, is straightforward so we won't review it in any detail here. The next two examples show the use of the number range class as a field in a JSON data holder class. The JsonObjNumRange class is used as a class field in the JsonObjIsOpCode and the JsonObjIsDirective classes respectively.

This indicates to us that some of the opcode and standard directive arguments require the use of the number range class to specify a valid range of values. Also, we've seen this before during our review of the Thumb-1 instruction set's data files.

## Class Review: JsonObjTxtMatch.java

The JsonObjTxtMatch class is an important part of the assembler project because it defines how to identify a string artifact and turn it into a token. I want to take a moment to briefly outline the process used by the GenAsm assembler.

1. Enter static main, process arguments.
2. Load specified assembly source code file.
3. Run preprocessor directives, update the assembly source code accordingly.
4. Run the Lexer and break the assembly source code down into pieces of text creating a list of artifacts.
5. Run the Tokenizer and scan each artifact to identify the type of text it contains, and the type of token that will be used to represent it, using the text match rules defined in the `is_entry_types.json` data file.
6. Run the assembler and generate a binary representation of each token line.
7. Run the linker and finalize the binary program's structure before writing the final output file.

There's a lot going on, needless to say, an assembler is a fairly complex piece of software. Don't worry though you're almost finished with this class review and, including all the material we've covered up to this point, you should have a much better understanding of the project, its structure, and its classes. Let's take a look at the `JsonObjTxtMatch` class now.

*Listing 10-36. JsonObjTxtMatch.java Full Class Review*

```

01 package net.middlemind.GenAsm.JsonObjs;
02
03 import java.util.List;
04 import net.middlemind.GenAsm.Logger;
05
06 public class JsonObjTxtMatch extends JsonObjBase {
07     public static String special_wild_card = "*";
08     public static String special_end_line = "endl";
09     public static String special_range = "~";
10     public static String special_lowercase_range = "a~z";
11     public static String special_uppercase_range = "A~Z";
12     public static String special_lowercase_num_range = "a~z0~9";
13     public static String special_uppercase_num_range = "A~Z0~9";
14     public String obj_name = "JsonObjTxtMatch";
15     public List<String> starts_with;
16     public List<String> contains;
17     public List<String> ends_with;
18     public List<String> must_contain;

```

```

19     public List<String> must_not_contain;
20
21     @Override
22     public void Print() {
23         Print("");
24     }
25
26     @Override
27     public void Print(String prefix) {
28         super.Print(prefix);
29         Logger.wrl(prefix + "ObjName: " + obj_name);
30
31         Logger.wrl(prefix + "StartsWith:");
32         for (String s : starts_with) {
33             Logger.wrl(prefix + "\t" + s);
34         }
35
36         Logger.wrl(prefix + "Contains:");
37         for (String s : contains) {
38             Logger.wrl(prefix + "\t" + s);
39         }
40
41         Logger.wrl(prefix + "MustContain:");
42         for (String s : must_contain) {
43             Logger.wrl(prefix + "\t" + s);
44         }
45
46         Logger.wrl(prefix + "MustNotContain:");
47         for (String s : must_not_contain) {
48             Logger.wrl(prefix + "\t" + s);
49         }
50
51         Logger.wrl(prefix + "EndsWith:");
52         for (String s : ends_with) {
53             Logger.wrl(prefix + "\t" + s);
54         }
55     }
56 }
```

A complete listing of the JsonObjTxtMatch class.

Much of the class follows the same pattern we've seen time and time again with JSON data holder classes. We'll focus our attention on the class fields. For the first time in the class review for this package we've encountered a bit more complexity. In this case we have a number of static fields that are used to define special strings.

The first static field listed, line 7, is the `special_wild_card` field and it's used to hold the string representation of the wild card string. In this case an asterisk, `'*'`. The next field, `special_end_line`, specifies an end line string and the `special_range` field is used to specify the string used to split ranges on.

Ranges, in this regard are specially formatted strings that specify groups of characters. The first such static field is the `special_lowercase_range` field on line 10. It has a value of `"a~z"` and is used to indicate all lowercase characters in a text match object. Notice that because we use the dash, `'-'`, in numeric expressions and keeping with the tradition of avoiding any overlap in character use we decided to go with the tilde.

Take a moment to review the remaining static class fields. They define upper and lowercase versions of the supported ranges. Note the double range, `"a~z0~9"`, is used to indicate an alphanumeric value. Also note that when case sensitivity is important, we have upper- and lower-case versions of the string patterns.

The remaining class fields are simple and direct. They define rules used to match text. Each entry specifies a list of strings that are validated in the way described by the field name. The `must_contain` and `must_not_contain` fields do not support ranges. So, you can't specify `"a~z"` as one of the entries in the `must_contain` string list, it won't be processed as a range.

That brings us to the conclusion of this section. I wanted to mention we could have just used Java supported regex to handle text matches, but I wanted to implement everything in the project as bare bones as possible, with the exception of the JSON parsing library, Gson. It didn't make sense to write up a JSON parser on top of all the other challenges on this project. In the next section we'll take a look at the `JsonObjTxtMatch` class in action.

## Demonstration: JsonObjTxtMatch.java

To demonstrate the `JsonObjTxtMatch` class we'll take a look at a few examples from various classes in the project.

*Listing 10-37. JsonObjTxtMatch.java Demonstration*

File: `Utils.java`

Method: `GetIntsFromRange`

Snippet:

```
1 return GetIntsFromRange(range, JsonObjTxtMatch.special_range);
```

File: `JsonObjIsEntryType.java`

Method: n/a

Snippet:

```
1 public JsonObjTxtMatch txt_match;
```

File: `TokenerThumb.java`

Method: `LineTokenize`

Snippet:

```
1 } else if(withStarts.equals(JsonObjTxtMatch.special_lowercase_range)) {  
2     //Found lower case character range  
3     char lc = payload.charAt(0);  
4     if(Character.isLowerCase(lc)) {  
5         withStarts = (lc + "");  
6         withStartsLen = 1;  
7         lfound = true;  
8         break;  
9     }  
10 }
```

A few examples of the `JsonObjTxtMatch` class in use.

The first example shown in the previous listing is from the `Utils` class' `GetIntsFromRange` method. This is a simple example where the utility method is overloaded to provide a default value for the range delimiter, `special_range` field of the `JsonObjTxtMatch` class. The next example is also very simple. This one is from the `JsonObjIsEntryType` class. One of the key fields in this class is an instance of the `JsonObjTxtMatch` class. As

we've mentioned earlier it's used to match text and identify it as a certain token type.

The last example shown is from the `TokenerThumb` class' `LineTokenize` method. This class is a tokenizer implemented for the Thumb-1 instruction set but you can probably use it for many other instruction sets. In line 1 if the `withStarts` variable is the string, `special_lowercase_range`, then we check the `payload`, `string` we're tokenizing, on lines 3 – 9. That's all we'll say about this snippet of code now. We'll cover this class in more detail later on in the text. And with that we have finished the review of the `JsonOBbjs` package. No small feat. That brings us to a good stopping point. There is still a lot of code for us to process at this stage in the review. We'll pick things back up in Chapter 11 with a review of the `Lexer` classes.

## Chapter Conclusion

In this chapter we took a look at a lot of classes. Building on top of your experience with the ARM Thumb-1 instruction set, the preprocessor directives, the opcodes, the standard directives, and the JSON data files. We've now completed part 1 of the base class review. We have a bit more code to get through in Chapter 11, part 2 of the base class review. Let's take a look at the classes we've covered in this chapter.

- **Assemblers Package**
  - **Assembler.java:** An interface that defines the minimum requirements necessary to implement an `Assembler`.
  - **AssemblerEventHandler.java:** An interface that defines how to implement an assembler event handler callback class.
  - **Symbols.java:** The `Symbols` class is used to hold the `Symbol` objects defined by a given assembly program.
  - **Symbol.java:** The `Symbol` class holds information like the absolute line number, the active line number, and in some cases

the value associated with a symbol definition.

- **Exceptions Package**
  - **ExceptionBase.java**: The super class to be used by all GenAsm project exception classes.
- **JsonObjs Package (Holder Classes)**
  - **JsonObj.java**: An interface that defines the minimum set of requirements to define a holder class.
  - **JsonObjBase.java**: A super class used by holder classes that provides an implementation of the `JsonObj` interface.
  - **JsonObjBitRep.java**: An implementation of a holder class designed to hold bit representation JSON sub-objects.
  - **JsonObjBitSeries.java**: An implementation of a holder class designed to hold bit series JSON sub-objects.
  - **JsonObjBitShift.java**: An implementation of a holder class designed to hold bit shift JSON sub-objects.
  - **JsonObjIlsSets.java**: An implementation of a holder class designed to hold instruction set definitions.
  - **JsonObjIlsSet.java**: An implementation of a holder class designed to hold the instruction set sub-object.
  - **JsonObjIlsFile.java**: An implementation of a holder class designed to hold the instruction set file sub-object.
  - **JsonObjLineHexReps.java**: A holder class that contains hex representation line sub-objects.
  - **JsonObjLineHexRep.java**: A holder class that contains hex representation line data.

- **JsonObjNumRange.java:** A holder class that contains number range sub-object data.
- **JsonObjTxtMatch.java:** A holder class that contains text match sub-object data.

It can be difficult to visualize, or otherwise absorb this much information so don't be discouraged if you don't feel like you have it down completely. Take the time to review the chapter again if need be. I'm sure you'll find that this chapter provides a solid reference and example of the first part of project's base classes. We're not out of the woods quite yet. We still have to review the following packages:

1. Lexers
2. Linkers
3. Loaders
4. PreProcessors
5. Tokeners

Once we're done with them, we'll have a solid foundation to build on and in fact, we'll be reviewing the implementation of the Thumb-1 instruction set in coming chapters. At the conclusion of that review step, we'll then be able to start working with the included example assembly programs.



# **Chapter 11: The Base Classes Part 2**

In this chapter we'll pick up the review of the base classes where we left off in part 1, Chapter 10, with the review of the classes that power the lexer. We have a few very important base classes left to review so let's get to it!

## **Package Review: Lexers**

The next package up for review is the Lexers package. This package holds the base classes that define how to implement a lexer that will plug into the GenAsm assembler's process. There are a few supporting classes that are used to describe the information that the lexer creates. Let's take a look at some code.

## **Class Review: Artifact.java**

The first class that we'll review from the Lexers package is the `Artifact` class. This class is used to represent a piece of text extracted from a line of assembly source code.

*`Listing 11-1. Artifact.java Full Class Review`*

```
01 package net.middlemind.GenAsm.Lexers;  
02  
03 public class Artifact {  
04     public String obj_name = "Artifact";  
05     public int posStart;  
06     public int posStop;  
07     public int len;  
08     public int lineNumber;  
09     public int index;  
10     public String payload;  
11 }
```

A complete listing of the `Artifact` class.

The class is surprisingly simple and definitely powerful. As you might expect the `obj_name` field is present and defaulted to the name of the class. This class is a bit different from the JSON data holder classes we've reviewed earlier in that it isn't driven by JSON data instead it's driven by the lexer process. This class does show up in JSON output files generated by the assembler, so we've kept the `obj_name` field convention to help identify the object in JSON output files.

The next field, `posStart`, is used to indicate the starting position on a line of assembly code represented by a string. The subsequent field, `posStop`, is used to indicate the stop position of the sub-string that this `Artifact` represents. Next the `len` field indicates the length of the sub-string while the `lineNum` field is the line number of the assembly source code that we're currently processing.

The `index` field is an integer that indicates the position of this artifact in the series of artifacts extracted from the source code line. You can think of it as an array index where all the artifacts on a given line take a position in the array. The last field, `payload`, is an important one. It holds a copy of the extracted sub-string. In the next section we'll take a look at the class in action.

## Demonstration: Artifact.java

There are a few demonstrations of the `Artifact` class listed subsequently. Take a look at them and try to imagine the class in use while doing so. Remember the purpose of this class is to hold pieces of text from the original assembly source code that may correspond with a token. We don't always know at this point if we've encountered something valid but we can identify the potential pieces of text none-the-less. This process is handled by the lexer, creation of tokens is taken care of by the tokenizer. Let's take a look at some examples.

*Listing 11-2. Artifact.java Demonstration*

File: Token.java

Method: n/a

Snippet:

```
1 public Artifact artifact;
```

File: ArtifactLine.java

Method: n/a

Snippet:

```
1 public List<Artifact> payload;
```

File: LexerThumb.java

Method: LineLexelize

Snippet:

```
01 } else if(Contains(CHAR_GROUP_START, chars[i])) {  
02     artifact = new Artifact();  
03     artifact.lineNum = lineNumber;  
04     artifact.posStart = i;  
05     artifact.posStop = i;  
06     artifact.len = 1;  
07     artifact.index = count;  
08     artifact.payload = "" + chars[i];  
09     ret.payload.add(artifact);  
10     count++;  
11     artifact = null;  
12  
13 }
```

File: TokenerThumb.java

Method: LineTokenize

Snippet:

```
01 for(Artifact art : line.payload) {  
02     current = art;  
03     found = false;  
04     payload = art.payload;  
05     ...  
06 }
```

A few examples of the Artifact class in use.

The first example is a class field from the `Token` class. This tells us that once that artifact is identified, and a token is created, the original artifact object is kept in the class' `artifact` field. It helps to be able to peek inside a token object and see what artifact was used to drive the creation of that token. The next example is from the `ArtifactLine` class. This class is used to represent all of the artifacts found on one line of assembly source code. As such it needs a place to store those artifacts, the `payload` field of the class is a list of `Artifact` objects from a lexerized line of assembly source code.

The third example shown in the previous listing is from the `LexerThumb` class' `LineLexerize` method. The `LexerThumb` class is the implementation of the lexer for the Thumb-1 instruction set. In this example we're handling the case where the character we're checking is the group start character, line 1. On lines 2 – 8 the artifact representing the group start is created. Then on line 9 it's added to the set of artifacts extracted from the line of assembly source code.

In the last example for this class, we are looking at a code snippet from the `TokenerThumb` class' `LineTokenize` method. The tokener's job is to identify artifacts as being of a specific type. In the snippet of code listed previously we are looping over the contents of an `ArtifactLine`'s `payload`, a list of `Artifact` objects. The purpose of this example is to demonstrate that the line tokenizer works by processing all of the identified artifacts in an artifact line, also referred to as a line of assembly source code.

## Class Review: ArtifactLine.java

The `ArtifactLine` class is used to represent a line of assembly source code and all the associated, extracted, artifacts. This is similar in nature to the JSON holder classes and their plural, singular naming convention, except in this case we have the `Artifact` and `ArtifactLine` classes with the `Artifact` classes acting as a sub-object to the `ArtifactLine` class which holds a list of artifact identified on a given line of assembly source code.

*Listing 11-3. ArtifactLine.java Full Class Review*

```
01 package net.middlemind.GenAsm.Lexers;
02
03 import java.util.List;
04
05 public class ArtifactLine {
06     public String obj_name = "ArtifactLine";
07     public int lineNumber;
08     public int sourceLen;
09     public String source;
10     public List<Artifact> payload;
11 }
```

A complete listing of the `ArtifactLine` class.

Let's quickly review the class' fields and get an idea of how the class is used. The ubiquitous `obj_name` field is the first entry on line 6. The value of the class field is set to the name of the class. The next field, `lineNum`, is used to represent the line number that this `ArtifactLine` represents. Subsequently, we have the `sourceLen` field which indicates the length of the string held in the `source` field. The `source` field in this case is the original assembly source code line while the `payload` field holds all the `Artifacts` that were extracted from this line.

## Demonstration: ArtifactLine.java

The `ArtifactLine` class has a few examples for us to look at. Take a moment to look at the subsequent examples before we review them.

*Listing 11-4. ArtifactLine.java Demonstration*

File: Lexer.java

Method: n/a

Snippet:

```
01 public List<ArtifactLine> FileLexerize(List<String> file);
```

File: AssemblerThumb.java

Method: n/a

Snippet:

```
01 public List<ArtifactLine> asmDataLexed;
```

File: TokenLine.java

Method: n/a

Snippet:

```
01 public ArtifactLine source;
```

File: Tokener.java

Method: n/a

Snippet:

```
01 public TokenLine LineTokenize(ArtifactLine line, int lineNumber, JsonObject entryTypes) throws ExceptionNoTokenerFound;
02 public List<TokenLine> FileTokenize(List<ArtifactLine> file, JsonObject entryTypes) throws ExceptionNoTokenerFound;
```

File: TokenerThumb.java

Method: FileTokenize

Snippet:

```
01 public List<TokenLine> FileTokenize(List<ArtifactLine> file, JsonObject entryTypes) throws ExceptionNoTokenerFound {
02     int lineNumber = 0;
03     ArrayList<TokenLine> ret = new ArrayList<>();
04     for(ArtifactLine art : file) {
05         ret.add(LineTokenize(art, lineNumber, entryTypes));
06         lineNumber++;
07     }
08     return ret;
09 }
```

A few examples of the ArtifactLine class in use.

The `ArtifactLine` class is an integral part of the assembler's process and as such you'll see it appear in a number of different places. The first example is from the `Lexer` interface's `FileLexelize` method signature. Notice that when lexing a file the method takes a list of strings as the assembly source code and returns a list of `ArtifactLine` instances after running the lexer.

Another example, the second one shown in the previous listing, is from the `AssemblerThumb` class' fields. The `asmDataLexed` field is used to hold lexed assembly source code lines. This shows that the lexing process is run as part of the assembly process. In the third example shown we can see that the `TokenLine` class has a class field, `source`, that is an instance of an `ArtifactLine` class.

This indicates to us that as the objects are transformed from lower-level string wrappers, like the artifact classes, to tokenized objects we maintain the connection to the lower-level objects by carrying them along with the process. In this case we can see that a `TokenLine` object contains a reference to the `ArtifactLine` object that it is based on.

Keep an eye out for connections of this kind as we review these classes. The fourth example shown is from the `Tokener` interface and it demonstrates the signature of the two methods required to properly define an implementation of the `Tokener` interface that plugs into the `GenAsm` assembly process.

There's some important knowledge hidden in these method signatures. Let's take a look at them. In the first method signature of this example, `LineTokenize`, notice that among other arguments it takes an `ArtifactLine` object instance and returns a `TokenLine` object instance. The nature of the relationship between the two classes can be seen clearly here.

It's further expressed in the next method signature from this example, the `FileTokenize` method. Notice that this method takes a `List` of `ArtifactLines` as an argument, among others, and returns a `List` of `TokenLines`. Now we can see that at the file level we're simply working with lists of the operative classes. First identifying certain artifacts then converting those artifacts to tokens all the while maintaining the integrity of the original

line of assembly source code by maintaining a connection from tokens to artifacts to the original line of assembly source code.

In the fifth, and last example, shown in the previous listing we're looking at the `TokenerThumb` class' `FileTokenize` method. Take a moment to realize that this example is an implementation of the interface example we looked at previously. The main take away from this example is that the method takes a `List` of `ArtifactLine` objects and returns a `List` of `TokenLine` objects. To this end we iterate over all the `ArtifactLines` and build up a list of `TokenLines` as shown on lines 4 – 7 of this example's code snippet.

## Interface Review: `Lexer.java`

In this section we'll review the `Lexer` interface. The interface defines the minimum requirements necessary to implement a `Lexer` class that plugs into the `GenAsm` assembler. Take a look at the full interface shown subsequently.

*Listing 11-5. `Lexer.java` Full Interface Review*

```
01 package net.middlemind.GenAsm.Lexers;
02
03 import java.util.List;
04
05 public interface Lexer {
06     public ArtifactLine LineLexerize(String line, int lineNumber);
07     public List<ArtifactLine> FileLexerize(List<String> file);
08 }
```

A complete listing of the `Lexer` interface.

The interface has some hidden information for us to discover. If you take a look at the method signatures you should notice that the same file-level, line-level method setup exists here as in the `Tokener` interface we looked at in a previous example. What's more the general flow of data can be assumed from the method arguments and return types.

In this case a file consists of a `List of Strings` and when processed, `lexerize`, returns a `List of ArtifactLine` objects. Similarly, the `LineLexerize` method signature shows a string argument `line`, and a return type of `ArtifactLine`. From the structure of the interfaces, we've reviewed in this chapter, you can discern the flow of information as defined by the assembly process. I'll detail that flow here.

<b>Input</b>	<b>Process</b>	<b>Output</b>
(Input Files)	Static Main	<code>List&lt;String&gt;</code>
<code>List&lt;String&gt;</code>	PreProcessor	<code>List&lt;String&gt;</code>
<code>List&lt;String&gt;</code>	Lexer	<code>List&lt;ArtifactLine&gt;</code>
<code>List&lt;ArtifactLine&gt;</code>	Tokener	<code>List&lt;TokenLine&gt;</code>
<code>List&lt;TokenLine&gt;</code>	Assembler	<code>List&lt;TokenLine&gt;</code>
(with binary representation)		
<code>List&lt;TokenLine&gt;</code>	Linker	(Output Files)

Take a look at the previously listed chart and make sure that you understand the flow of data shown. This information can come in handy when the concepts and names of things are spinning around in your head. I find that reading through it is very grounding. You can almost see the process happen as your read through it. In the next section we'll take a look at the `Lexer` interface in use.

## Demonstration: `Lexer.java`

The `Lexer` class is responsible for converting the assembly source code text into artifacts, or pieces of the source string that we'll identify later on during tokenization. There is only one place in the project where we use the `Lexer` interface, during its implementation for the Thumb-1 instruction set. Let's take a look.

### *Listing 11-6. `Lexer.java` Demonstration*

File: `LexerThumb.java`

Method: `LexerThumb`

Snippet:

```
01 public class LexerThumb implements Lexer {
02     public ArrayList<ArtifactLine> FileLexerize(List<String> file) {
```

```
03     ...
04 }
05
06 public ArtifactLine LineLexerize(String line, int lineNumber) {
07     ...
08 }
09 }
```

An example of the Lexer interface in use.

In the previously shown example we see the Lexer interface in use through the implementation of the interface for the ARM Thumb-1 instruction set. There's not much for us to talk about here. The details of the class methods will be reviewed, in detail, in a subsequent chapter. The main thing to note is how the Interfaces and Classes work together to create an extensible coding model.

The structure of the project allows for you to define your own implementation of the `Lexer` interface, or any other interface for that matter. You'll find that adding new implementations of existing interfaces to the project is trivial compared to coding in entirely new and foreign classes.

## Package Review: Linkers

The next package for use to review is a short one. The Linkers package only has one entry the `Linker` interface. This is due to the fact that this class runs after all other processing classes. As such all the objects needed to define the problem already exist in other packages. Thus, this general package only has one interface. Let's take a look.

### Interface Review: Linker.java

The `Linker` interface defines the minimum set of methods necessary to implement a linker and plug it into the GenAsm assembler's process. The linker is responsible for connecting different areas of an assembly source code file. More robust assemblers will have more work to do but our assembler is a bit less complex so we really only have to worry about properly connecting and aligning the code and data areas taking into account their line

numbers and ordering things properly. We'll have a clearer picture of this interface's use as we review classes that implement it for the ARM Thumb-1 instruction set, a little later on in the text.

***Listing 11-7. Linker.java Full Interface Review***

```
01 package net.middlemind.GenAsm.Linkers;  
02  
03 import net.middlemind.GenAsm.AssemblersAssembler;  
04  
05 public interface Linker {  
06     public void RunLinker(Assembler assembler, String  
assemblySourceFile, String outputDir, Object otherObj, boolean verbose,  
boolean quellOutput) throws Exception;  
07 }
```

A complete listing of the Linker interface.

The `Linker` interface defines one method, the `RunLinker` method. Let's quickly review the method's arguments. The first argument, `assembler`, is a reference to the active assembler. The second argument, `assemblySourceFile`, is the full path to the assembly source file. The third argument, `outputDir`, is a string that indicates the target output directory to write the final output files.

The subsequent argument, `otherObj`, is a general Java object that can be used to customize the process for a specific instruction set implementation. The remaining method arguments are Boolean flags that indicate if verbose logging should be turned on and if the output files should be quelled, respectively.

## **Demonstration: Linker.java**

To demonstrate the `Linker` class in action we have a few examples to look at, listed subsequently.

***Listing 11-8. Linker.java Demonstration***

File: GenAsm.java

Method: n/a

Snippet:

```
01 public static Linker ASM_LINKER = null;
```

File: GenAsm.java

Method: main

Snippet:

```
01 cTmp = Class.forName(ASM_LINKER_CLASS);
02 ASM_LINKER = (Linker)cTmp.getDeclaredConstructor().newInstance();
```

File: LinkerThumb.java

Method: n/a

Snippet:

```
01 public class LinkerThumb implements Linker {
01     public boolean verboseLogs = false;
02     public boolean quellFileOutput = false;
03     public ArrayList<Byte> binBe = null;
04     public ArrayList<Byte> binLe = null;
05     public Hashtable<String, String> hexMapBe = null;
06     public Hashtable<String, String> hexMapLe = null;
07
08     @Override
09     public void RunLinker(Assembler assembler, String
assemblySourceFile, String outputDir, Object otherObj, boolean verbose,
boolean quellOutput) throws Exception {
10         ...
11     }
12 }
```

An example of the Linker interface in use.

The first example is from the `GenAsm` class, the static main entry point of the project. Notice that there is a field to hold a reference to the currently loaded linker, `ASM_LINKER`. In the next example, also from the `GenAsm` class, in the class' `main` method. On line one we load up an instance of the `Linker` class dynamically using the full package name of the class.

And on line 2 of the example the class constructor is called, and the class is cast up to the super class. We need to cast up to the super class to

preserve generality and allow extending classes to be used. The developer can implement things as they need and use extending classes where needed in a specific assembler implementation. The third example is from the `LinkerThumb` class' implementation of the `Linker` interface.

The main takeaway from this example is the implementation of the interface and the overridden, required, method `RunLinker`. That brings us to the conclusion of the `Linkers` package review. In the next section we'll look into the `Loaders` package.

## Package Review: Loaders

The `Loaders` package is used to hold classes involved with the loading of JSON data files into holder classes. We've seen a number of holder classes in this chapter, but we haven't seen too many loader classes. This package contains a few of the base loader classes used in the GenAsm assembler project.

### Interface Review: Loader.java

The first topic for us to review is the `Loader` interface. This interface defines the minimum set of requirements needed to implement a loader class that can plug into the GenAsm assembler's process.

*Listing 11-9. Loader.java Full Interface Review*

```
01 package net.middlemind.GenAsm.Loaders;
02
03 import net.middlemind.GenAsm.Exceptions.Thumb.ExceptionLoader;
04 import net.middlemind.GenAsm.JsonObjs.JsonObj;
05
06 public interface Loader {
07     public JsonObj ParseJson(String json, String targetClass, String
fileName) throws ExceptionLoader;
08 }
```

A complete listing of the `Loader` interface.

The interface is very concise, so we won't have much to talk about. Let's take a look at the `ParseJson` method's arguments. The first argument is a string containing a JSON object to parse. The second argument is the full package name of the target class the JSON object should be loaded into. The last argument is a string that holds the path to the JSON file that we're trying to load. There's not much more to discuss here so let's move on. In the next section we'll take a look at a demonstration of the `Loader` interface in action.

## Demonstration: Loader.java

Let's take a look at a few places in the code where the `Loader` interface is used. Keep in mind that for each JSON file we want to load into an assembler implementation we need a new class that implements the `Loader` interface.

*Listing 11-10. Loader.java Demonstration*

File: AssemblerThumb.java

Method: n/a

Snippet:

```
01 public Map<String, Loader> isaLoader;
```

File: AssemblerThumb.java

Method: LoadAndParseJsonObjData

Snippet:

```
01 cTmp = Class.forName(entry.loader_class);
02 ldr = (Loader)cTmp.getDeclaredConstructor().newInstance();
03 json = null;
04 jsonName = null;
05 jsonObj = null;
06
07 isaLoader.put(entry.loader_class, ldr);
```

File: LoaderIsTestFileVals.java

Method: LoaderIsTestFileVals

Snippet:

```
01 public class LoaderIsTestFileVals implements Loader {
02     public String obj_name = "LoaderIsTestFileVals";
03
04     @Override
05     public JsonObjectIsTestFileVals ParseJson(String json, String
targetClass, String fileName) throws ExceptionLoader {
```

```
06      ...
07  }
08 }
```

An example of the Loader interface in use.

The first example is from the `AssemblerThumb` class. The `isaloader` class field is a `Map` between a string name and an instance of the `Loader` class, an implementation of the `Loader` interface. It's used to store different loader classes by name. The second example is also from the `AssemblerThumb` class and demonstrates the `isaloader` field in use. The snippet of code associated with this example is from the `AssemblerThumb` class' `LoadAndParseJsonObjData` method.

The last example in the previous listing is from the `LoaderIsTestFileVals` class and it is an example of a class that implements the `Loader` interface. Notice that this class only adds a field, `obj_name`, that you should be familiar with. Also notice that the interface's method definition, `ParseJson`, is overridden by the implementing class as required. In the next few sections, we'll take a look at a few loader classes that implement the `Loader` interface we reviewed here.

## Class Review: `LoaderBitSeries.java`

The `LoaderBitSeries` class is a loader class that is designed to load a bit series JSON object. This is the `bit_series` sub-object that we've encountered before during the review of the holder classes, Chapter 10, and the JSON data files from earlier in the text.

*Listing 11-11. `LoaderBitSeries.java` Full Class Review*

```
01 package net.middlemind.GenAsm.Loaders;
02
03 import com.google.gson.Gson;
04 import com.google.gson.GsonBuilder;
05 import java.lang.reflect.InvocationTargetException;
06 import net.middlemind.GenAsm.Exceptions.Thumb.ExceptionLoader;
07 import net.middlemind.GenAsm.JsonObjs.JsonObjBitSeries;
```

```

08
09 public class LoaderBitSeries implements Loader {
10     public String obj_name = "LoaderBitSeries";
11
12     @Override
13     public JsonObjBitSeries ParseJson(String json, String targetClass,
14                                     String fileName) throws ExceptionLoader {
14         GsonBuilder builder = new GsonBuilder();
15         builder.setPrettyPrinting();
16
17         Gson gson = builder.create();
18         try {
19             JsonObjBitSeries jsonObj =
20                 (JsonObjBitSeries)Class.forName(targetClass).getConstructor().newInstance(
21 );
22             jsonObj = gson.fromJson(json, jsonObj.getClass());
23             jsonObj.name = targetClass;
24             jsonObj.fileName = fileName;
25             jsonObj.loader = getClass().getName();
26             return jsonObj;
27         } catch (ClassNotFoundException | NoSuchMethodException |
28 InstantiationException | IllegalAccessException |
29 InvocationTargetException e) {
26             throw new ExceptionLoader("Could not find target class, " +
targetClass + ", in loader " + getClass().getName());
27         }
28     }
29 }
```

A complete listing of the LoaderBitSeries class.

On line 10 of the previous listing, you can see the class field `obj_name` and its default value, the name of the class. This approach follows the convention we've seen thus far with the JSON data holder classes. The JSON object is loaded into a holder class on lines 19 – 20 and the class' base fields are set on lines 20 – 23. This pattern, although simple, is actually quite durable and you'll see it used over and over again for each of the loader classes we review.

The `LoaderBitSeries` class is not used in the project as is. The reason for this is that we're not loading any bit series JSON files directly that would require this loader. The distinction here is that we're not talking about loading a JSON bit series object, these are loaded as a part of a larger JSON object like an opcode. Rather, we aren't loading a JSON text file of bit series objects. The intended use of this class is to test loading bit series sub-objects to ensure that they load fine when they're part of a larger JSON object. As such we won't demonstrate this class and instead, we'll move on to the next class review.

## Class Review: `LoaderLineHexReps.java`

In this section we'll review the `LoaderLineHexReps` class. This class is used to load JSON data into a holder class designed to store hexadecimal representations of a program's instruction at a given memory address in a ARM Thumb-1 binary file. The purpose of this class is to load up a JSON data file with a series of hexadecimal values, one for each memory address, and compare it to a known set of values, for a given assembled program. This class is used in the project's unit tests as part of the program verification checks.

*Listing 11-12. LoaderLineHexReps.java Full Class Review*

```
01 package net.middlemind.GenAsm.Loaders;
02
03 import com.google.gson.Gson;
04 import com.google.gson.GsonBuilder;
05 import java.lang.reflect.InvocationTargetException;
06 import net.middlemind.GenAsm.Exceptions.Thumb.ExceptionLoader;
07 import net.middlemind.GenAsm.JsonObjs.JsonObjLineHexReps;
08
09 public class LoaderLineHexReps implements Loader {
10     public String obj_name = "LoaderLineHexReps";
11
12     @Override
13     public JsonObjLineHexReps ParseJson(String json, String targetClass,
14                                         String fileName) throws ExceptionLoader {
15         GsonBuilder builder = new GsonBuilder();
16         builder.setPrettyPrinting();
```

```

17     Gson gson = builder.create();
18     try {
19         JsonObjLineHexReps jsonObj =
(JsonObjLineHexReps)Class.forName(targetClass).getConstructor().newInstance();
20         jsonObj = gson.fromJson(json, jsonObj.getClass());
21         jsonObj.name = targetClass;
22         jsonObj.fileName = fileName;
23         jsonObj.loader = getClass().getName();
24         return jsonObj;
25     } catch (ClassNotFoundException | NoSuchMethodException |
InstantiationException | IllegalAccessException | InvocationTargetException e) {
26         throw new ExceptionLoader("Could not find target class, " +
targetClass + ", in loader " + getClass().getName());
27     }
28 }
29 }
```

A complete listing of the LoaderLineHexReps class.

The LoaderLineHexReps class is very similar to the LoaderBitSeries class we've just reviewed. The only difference is in the type of data being processed by the JSON data loader class. The holder class is referenced on lines 13 and 19. The `obj_name` field is used to provide a name for the class in the same way that it's used to provide names for the JSON holder classes. There isn't more to discuss regarding this class. This class also doesn't have a demonstration because it is only used to test loading up line hex representations to verify it works and can be used in the project's unit tests.

## Class Review: LoaderIsSets.java

The LoaderIsSets class is a very important one. It's used to load up the `is_sets.json` data file which contains all the instruction set definitions and their associated files. Note that sub-objects are loaded by Gson and a separate loader class is not needed for them. Loader classes associate to JSON text files not JSON objects.

*Listing 11-13. LoaderIsSets.java Full Class Review*

```
01 package net.middlemind.GenAsm.Loaders;
02
03 import com.google.gson.Gson;
04 import com.google.gson.GsonBuilder;
05 import java.lang.reflect.InvocationTargetException;
06 import net.middlemind.GenAsm.Exceptions.Thumb.ExceptionLoader;
07 import net.middlemind.GenAsm.JsonObjs.JsonObjIsFile;
08 import net.middlemind.GenAsm.JsonObjs.JsonObjIsSet;
09 import net.middlemind.GenAsm.JsonObjs.JsonObjIsSets;
10 import net.middlemind.GenAsm.Logger;
11
12 public class LoaderIsSets implements Loader {
13     public String obj_name = "LoaderIsSets";
14
15     @Override
16     public JsonObjIsSets ParseJson(String json, String targetClass,
17         String fileName) throws ExceptionLoader {
18         Logger.wrl("LoaderIsSets: ParseJson");
19         GsonBuilder builder = new GsonBuilder();
20         builder.setPrettyPrinting();
21
22         try {
23             JsonObjIsSets jsonObj =
24                 (JsonObjIsSets)Class.forName(targetClass).getConstructor().newInstance();
25             jsonObj = gson.fromJson(json, jsonObj.getClass());
26             jsonObj.name = targetClass;
27             jsonObj.fileName = fileName;
28             jsonObj.loader = getClass().getName();
29
30             for(JsonObjIsSet entry : jsonObj.is_sets) {
31                 entry.name = entry.getClass().getName();
32                 entry.fileName = fileName;
33                 entry.loader = getClass().getName();
34
35                 for(JsonObjIsFile fentry : entry.is_files) {
36                     fentry.name = fentry.getClass().getName();
37                     fentry.fileName = fileName;
38                     fentry.loader = getClass().getName();
39                 }
40             }
41         } catch (Exception e) {
42             e.printStackTrace();
43         }
44     }
45 }
```

```

40         return jsonObj;
41     } catch (ClassNotFoundException | NoSuchMethodException |
42 InstantiationException | IllegalAccessException |
43 InvocationTargetException e) {
44         throw new ExceptionLoader("Could not find target class, " +
45 targetClass + ", in loader " + getClass().getName());
46     }
47 }

```

A complete listing of the `LoaderIsSets` class.

Even though the `LoaderIsSets` class is a very important loader class. It isn't any different from the loader classes we've seen thus far. Notice that on line 13 the class field `obj_name` has the same name as the class it is a member of. On lines 13 and 23 the holder class used by the loader is an instance of the `JsonObjIsSets` class. In the next section we'll take a look at the class in use.

## Demonstration: LoaderIsSets.java

To demonstrate the `LoaderIsSets` class we'll take a look at a few examples of the class in use as part of the assembly process. The following two examples are from the `GenAsm` class' `main` method.

*Listing 11-14. LoaderIsSets.java Demonstration*

File: `GenAsm.java`

Method: `main`

Snippet:

```
1 LoaderIsSets ldrIsSets;
```

File: `GenAsm.java`

Method: `main`

Snippet:

```
1 if(!Utils.IsEmpty(json)) {
2     try {
3         cTmp = Class.forName(ASM_SETS_LOADER_CLASS);
```

```
4     ldrIsSets =
5     (LoaderIsSets)cTmp.getDeclaredConstructor().newInstance();
6     ASM_SETS = ldrIsSets.ParseJson(json, ASM_SETS_TARGET_CLASS,
7     ASM_SETS_FILE_NAME);
8     ...
9 }
```

An example of the LoaderIsSets class in use.

The first example from the `GenAsm` class' `main` method shows the class in use as a local variable. This variable is used in the loading of the instruction set files. In the second example, also from the `GenAsm` class' `main` method, the snippet of code shows the loading of the `is_sets` JSON data file as part of the `GenAsm` class' `main` method.

On line 1 if the `json` variable is not empty then we create a new instance of the loading class specified, line 3, and on line 4 the loader class is initialized, and the instruction set files are loaded on line 5. That's all there is to the example. It's very important because it's the source of all the data files the assembler has access to. In the next section we'll take a look at the project's `PreProcessors` package and associated classes.

## Package Review: PreProcessors

The next package that we'll review in this chapter is the `PreProcessors` package. This package is very succinct. It contains only one interface, the `PreProcessor` interface which we'll review now.

### Interface Review: PreProcessor.java

The `PreProcessor` interface defines the minimum set of requirements necessary for a class to implement the `PreProcessor` interface and plug into the `GenAsm` assembly process. We'll list the interface' contents here.

*Listing 11-15. PreProcessor.java Full Interface Review*

```
1 package net.middlemind.GenAsm.PreProcessors;  
2  
3 import java.util.List;  
4  
5 public interface PreProcessor {  
6     public List<String> RunPreProcessor(String assemblySourceFile, String  
outputDir, Object other) throws Exception;  
7 }
```

A full listing of the `PreProcessor` interface.

The `PreProcessor` interface defines the minimum set of methods that need to be implemented in order to define a `PreProcessor` class that works with the `GenAsm` assembler. The main method defined by the interface is the `RunPreProcessor` method. Let's take a moment to talk about this method. The first argument is a string that points to the assembly source code file. The next argument is a string that specifies the output directory. Lastly, the third argument, `other`, is a general object that can be used to customize the preprocessor.

## Demonstration: `PreProcessor.java`

To demonstrate the `PreProcessor` directive, we've put together a few examples shown in the subsequent listing.

*Listing 11-16. `PreProcessor.java` Demonstration*

File: `PreProcessorThumb.java`

Method: n/a

Snippet:

```
1 public class PreProcessorThumb implements PreProcessor {}
```

File: `GenAsm.java`

Method: n/a

Snippet:

```
1 public static PreProcessor ASM_PREPROCESSOR = null;
```

File: `GenAsm.java`

Method: `main`

Snippet:

```
1 cTmp = Class.forName(ASM_PREPROCESSOR_CLASS);
2 ASM_PREPROCESSOR =
(PreProcessor)cTmp.getDeclaredConstructor().newInstance();
```

An example of the PreProcessor interface in use.

The first example is from the `PreProcessorThumb` class definition. Note that the class implements the `PreProcessor` interface by defining the `RunPreProcessor` method. The second example is from the `GenAsm` class definition. It shows the `ASM_PREPROCESSOR` static class field which is used to load up the preprocessor. The third example is from the `GenAsm` class' main method.

On line 1 the class is loaded and stored in the `cTmp` local variable. On line two the static class field is properly initialized and cast to the super class. The preprocessor is now instantiated, in a data driven way, and ready to use. That brings us to the conclusion of the `PreProcessors` package review. In the next section we'll start the review of the last package we need to cover, the `Tokeners` package.

## Package Review: Tokeners.java

The `Tokeners` package holds the base classes and interfaces that provide tokenization support. There is one main interface, the `Tokener` interface, and a few key classes – the `Token`, `TokenLine`, and `TokenSorter` classes. Let's take a look.

### Class Review: Token.java

The `Token` class is an important one. This class is used to represent a `Token` of a known type. This means that the `Token` is marked with an instance of the `JsonObjIsEntryType` class. Also, recall that `Tokens` are derived from `Artifacts` in the same way that `TokenLines` are derived from `ArtifactLines`. Take a moment to think about this. `Artifacts` are just sub-strings of text extracted from a line of text. `Tokens` are just objects that

now wrap the original Artifact and associate it with an `JsonObjIsEntryType`, `is_entry_type`, from the current instruction set.

*Listing 11-17. Token.java Full Class Review*

```
01 package net.middlemind.GenAsm.Tokeners;
02
03 import net.middlemind.GenAsm.Lexers.Artifact;
04 import java.util.List;
05 import net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsEntryType;
06 import net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsRegister;
07
08 public class Token {
09     public String obj_name = "Token";
10    public String type_name;
11    public String source;
12    public Object value;
13    public int lineNumAbs;
14    public int index;
15    public int payloadLen;
16    public int payloadArgLen;
17    public boolean isOpCodeArg;
18    public boolean isDirectiveArg;
19    public boolean isOpCode;
20    public boolean isDirective;
21    public boolean isComment;
22    public boolean isLabel;
23    public boolean isLabelRef;
24    public Artifact artifact;
25    public JsonObjIsEntryType type;
26    public JsonObjIsRegister register;
27    public List<Token> payload;
28 }
```

A complete listing of the Token class.

The Token class is a crucial one in our implementation of the Thumb-1 instruction set. It is used just about everywhere in the AssemblerThumb class as it's the primary object to interact with when assembling the source code. Let's take a look at the class' fields. As to be expected, the class has an

`obj_name` field that's defaulted to the name of the class. This follows the same pattern we've seen before in holder and loader classes.

The next entry is the `type_name` field and it holds a string representing the `JsonObjIsEntryType`, `is_entry_type`, of the token. The next field, `source`, is a string that holds the original source code that the `Token` is based on. The `value` field is used to hold an object that is associated with the `Token` as its value. This is mainly used with the `@EQU` standard directive.

The next field, `lineNumAbs`, is an integer that represents the original line number, from the source code, that this token is associated with. The `index` field is an integer that indicates the order of this `Token` in the `TokenLine`'s list of `Tokens`. The `payloadLen` field is an integer that represents the number of payload objects.

The `payloadArgLen` field is an integer that indicates the number of payload objects that are argument objects in this token's payload. The next seven Boolean fields are flags that indicate the type of the token object. The `isOpCodeArg` Boolean indicates if the `Token` is an opcode argument. Similarly, the `isDirectiveArg` field indicates if the token is a directive argument.

The next two Boolean flags, `isOpCode` and `isDirective`, are used to indicate if the `Token` is an opcode or a directive respectively. The next field is a Boolean flag that indicates if the `Token` is a comment while the remaining two Boolean fields indicate if the token is a label or a label reference.

The next class field, `type`, is a reference to the `JsonObjIsEntryType` instance that is named by the `type_name` field. The `register` field is used to hold a reference to the `JsonObjIsRegister` instance that is associated with the `Token`, if any. The last field in the class, line 27, is the `payload` field. This field is a `List of Tokens` and is used when a token has children. This occurs in the case of group, list, or comment tokens in a token line.

Take a moment to look over the class one more time. It's deceptively simple. This class is the workhorse of the assembly process, and you'll see it used throughout the `AssemblerThumb` class.

## Demonstration: Token.java

There are too many examples to use for the `Token` class, its use is ubiquitous in the `AssemblerThumb` class. So, I decided on one solid example that demonstrates the `Token` class as it is used in processing directives and their arguments.

*Listing 11-18. Token.java Full Class Review*

File: `AssemblerThumb.java`

Method: `PopulateDirectiveArgAndAreaData`

Snippet:

```
01 for(Token token : line.payload) {  
02     lastToken = token;  
03  
04     if(foundTtl &&  
token.type_name.equals(JsonObjIsDirectives.NAME_DIRECTIVE_TYPE_STRING)) {  
05         foundTtl = false;  
06         assemblyTitle = token.source;  
07  
08     }  
09     ...  
10 }
```

An example of the `Token` class in use.

The example shown in the previous listing is from the `AssemblerThumb` class' `PopulateDirectiveArgAndAreaData` method. On line 1 you can see that we're iterating over the contents of the `TokenLine` class' `payload` field. Each object in the data structure is an instance of the `Token` class, line 2. The code on line 4 indicates that a directive argument for the `@TTL` standard directive, has been found, line 4. If so, and the field is already set to true the `foundTtl` variable is then set to false, and the title string is stored in the `assemblyTitle` class field, lines 5 – 6. In the next section we'll take a look at the `TokenLine` class.

## Class Review: TokenLine.java

The `TokenLine` class is used to hold all the `Tokens` found on a line of assembly source code. The `TokenLine` class is akin to the `ArtifactLine` class we've reviewed earlier. As you can imagine `Token` and `TokenLine` classes are related in the same way `Artifact` and `ArtifactLine` classes are related.

*Listing 11-19. TokenLine.java Full Class Review*

```
01 package net.middlemind.GenAsm.Tokeners;
02
03 import net.middlemind.GenAsm.Lexers.ArtifactLine;
04 import net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsOpCode;
05 import net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsDirective;
06 import java.util.List;
07 import net.middlemind.GenAsm.Assemblers.Thumb.BuildOpCodeThumb;
08 import net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsValidLine;
09
10 public class TokenLine {
11     public String obj_name = "TokenLine";
12     public String payloadOpCode;
13     public String payloadDirective;
14     public String payloadBinRepStrEndianBig1;
15     public String payloadBinRepStrEndianLil1;
16     public String payloadBinRepStrEndianBig2;
17     public String payloadBinRepStrEndianLil2;
18     public int byteLength = 2;
19     public String addressHex;
20     public String addressBin;
21     public int addressInt;
22     public int lineNumberActive;
23     public int lineNumberAbs;
24     public int payloadLen;
25     public int payloadLenArg;
26     public boolean isLineEmpty;
27     public boolean isLineOpCode;
28     public boolean isLineDirective;
29     public boolean isLineLabelDef;
30     public ArtifactLine source;
31     public JsonObjIsValidLine validLineEntry;
32     public List<JsonObjIsOpCode> matchesOpCode;
```

```
33     public List<JsonObjIsDirective> matchesDirective;
34     public List<BuildOpCodeThumb> buildEntries;
35     public List<Token> payload;
36 }
```

A complete listing of the `TokenLine` class.

The `TokenLine` class is also a very important helper class that is used throughout the project and especially in the `AssemblerThumb` class. Let's take a look at the class' fields and get an understanding of how the class is used. The first field is the `obj_name` field, as you might have expected. The field is defaulted to the name of the class in the same way we've seen in previous class reviews.

The next field `payloadOpcode` is a string that's holds the name of the opcode found on this assembly source code line, if any. The `payloadDirective` is a class field that holds the name of the directive found on this assembly source code line, if any. The next four strings are used to hold binary representations of the assembly source code line for both little- and big-endian encodings. Also note that there are two lines of binary representations supported not just one.

This is due to certain branching opcodes that require two lines, 4 bytes of binary code. The subsequent class field, `byteLength`, indicates the total number of bytes needed to represent an opcode or directive. The following three fields, lines 19 – 21, are used to track the line number this `TokenLine` is associated with.

The `addressHex`, `addressBin`, and `addressInt` fields are all used to represent the line number that this `TokenLine` object is associated with but with the noted encoding, hex, binary, and decimal respectively. Following the address fields are two, line number fields, `lineNumActive`, `lineNumAbs`. The `lineNumActive` field represents only the lines of assembly source code that will be represented in the final program's binary output.

The `lineNumAbs` field represents every line of assembly source code whether it will be in the program's binary output or not. The next field

`payloadLen`, on line 24, tracks the number of objects in the payload. The `payloadLenArg` field, line 25, represents the number of argument objects in the payload. Following this field, on lines 26 – 29, are a series of Boolean flags similar to what we saw in the `Token` class.

Let's take a look at these Boolean fields. The `isLineEmpty` field is used to indicate if the line is considered empty. An empty line will not have an assembly source that ends up in the final binary representation of the assembly source file. The `isLineOpCode` and `isLineDirective` fields are Boolean flags that indicate if the line is an opcode or directive line. The last Boolean field in the set is the `isLineLabelDef` field and it's used to indicate if the line contains a label definition. The `source` field, line 30, is a reference to the `ArtifactLine` object that this `TokenLine` object is based on.

The `validLineEntry` field, line 31, is used to hold a reference to a valid line object instance that matches the pattern of this line of assembly source code. The `matchesOpCode` and `matchesDirective` fields are used to hold references to a list of possible opcode or standard directive matches, if any. These fields are only populated if the `TokenLine` class is an opcode or directive token line.

The next field, `buildEntries`, is used to hold the binary encoding results from processing each, active, line of assembly source code. The last field entry, line 35, is used to hold all the `Tokens` associated with the current line. That brings us to the conclusion of this class review. In the next section we'll take a look at the `TokenLine` class in action.

## Demonstration: `TokenLine.java`

To demonstrate the `TokenLine` class I've pulled a few of examples of the class in use. Take a look at the subsequent listing and try to imagine the `TokenLine` class in use as you read over them. We'll quickly review them before moving on to the next class up for review.

### *Listing 11-20. `TokenLine.java` Full Class Review*

File: `AssemblerThumb.java`

Method: `PopulateDirectiveArgAndAreaData`

Snippet:

```

01 asmAreaLinesData = new ArrayList<TokenLine>();
02 for(int z = areaThumbData.lineNumEntry + 1; z <
areaThumbData.lineNumEnd; z++) {
03     TokenLine line = asmDataTokened.get(z);
04     if(!line.isEmpty && !line.isLineLabelDef &&
line.isLineDirective) {
05         line.addressHex =
Utils.FormatHexString(Integer.toHexString(asmStartLineNumber +
activeLineCount), lineLenBytes);
06         line.addressBin =
Utils.FormatBinString(Integer.toBinaryString(asmStartLineNumber +
activeLineCount), lineBitSeries.bit_len);
07         line.addressInt = (asmStartLineNumber + activeLineCount);
08         line.lineNumActive = (line.addressInt/lineLenBytes);
09         asmAreaLinesData.add(line);
10         activeLineCount += lineLenBytes;
11     }
12 }
```

File: AreaThumb.java

Method: n/a

Snippet:

```

01 public TokenLine areaLine;
02 public TokenLine entryLine;
03 public TokenLine endLine;
```

File: Symbol.java

Method: n/a

Snippet:

```
01 public TokenLine line;
```

An example of the TokenLine class in use.

The first example, shown previously, is from the AssemblerThumb class' PopulateDirectiveArgAndAreaData method. On line 1 of the example snippet the local variable, `asmAreaLinesData`, is initialized. This is the variable that will collect all the `TokenLines` defined in the data area. The for loop on line 2 of the snippet iterates from the start of a data area, `areaThumbData.lineNumEntry`, to the end of the data area, `areaThumbData.lineNumEnd`.

Because we're working with absolute line numbers, we can pull the `TokenLine` object instance from the list of tokenized lines, `asmTokenedData`, and set the local `TokenLine` variable, `line`, on line 3 of the snippet. Next, on line 4, we check to see if the line we're working on is an active line. Active lines of assembly code are present in the final binary representation of the program while non-active lines are not.

If the `TokenLine` is an active line, then the `TokenLine`'s address fields are updated on lines 5 – 7 and take into consideration different numeric encodings. The `lineNumActive` field is updated to represent the current active line. Notice that the active line is added to the `asmAreaLinesData` list on line 9. The next example is from the `AreaThumb` class. It's a simple example that shows a few class field declarations. The fields are used to track which `TokenLine` certain key features occur on. In this case the `areaLine` field indicates which line that the area directive can be found on.

The same goes for the `entryLine` and `endLine` class fields. To further this notion let's look at the third example. This example is from the `Symbol` class. The class' `line` field is an instance of a `TokenLine` object and it represents the line that the current symbol is associated with. This wraps up the class demonstration section. In the next section we'll take a look at the last class up for review in this package, the `TokenSorter` class.

## Class Review: TokenSorter.java

The `TokenSorter` class, as you may have guessed, is used to sort a list of `Tokens`. Its implementation is simple and direct. Let's have a look.

*Listing 11-21. TokenSorter.java Full Class Review*

```
01 package net.middlemind.GenAsm.Tokeners;
02
03 import java.util.Comparator;
04
05 public class TokenSorter implements Comparator<Token> {
06     public enum TokenSorterType {
07         INDEX_ASC,
08         INDEX_DSC
09     }
```

```

10
11     public TokenSorterType sortType = TokenSorterType.INDEX_ASC;
12     public String obj_name = "TokenSorter";
13
14     public TokenSorter() {
15
16
17     public TokenSorter(TokenSorterType sType) {
18         sortType = sType;
19     }
20
21     @Override
22     public int compare(Token a, Token b) {
23         if(sortType == TokenSorterType.INDEX_ASC) {
24             return (a.index - b.index);
25         } else if(sortType == TokenSorterType.INDEX_DSC) {
26             return (b.index - a.index);
27         } else {
28             return (a.index - b.index);
29         }
30     }
31 }
```

A complete listing of the TokenSorter class.

The first thing to notice is that the class implements the `Comparator` interface on line 5 and that the interface is typed with the `Token` class. The class defines an enumeration on lines 6 – 9. The enumeration is used to customize the class' sorting method. It supports index based sorting ascending or descending. Direct your attention to the `sortType` class field on line 11, also notice that the `sortType` field is defaulted to index ascending. On lines 14 – 19 are the class constructors. Not much is going on here. We have one generic constructor and one that takes a sort type argument.

The class' operative function is the `compare` method on lines 22 – 30. Depending on the value of `sortType` the method returns an integer value that describes the order of the method's arguments, tokens `a` and `b`. That brings us to the conclusion of this section. Up next, we'll take a look at how the `TokenSorter` class is used by the assembler.

## Demonstration: TokenSorter.java

The `TokenSorter` class is best demonstrated with an example from the `AssemblerThumb` class.

*Listing 11-22. TokenSorter.java Demonstration*

File: `AssemblerThumb.java`

Method: `BuildBinOpCode`

Snippet:

```
1 Collections.sort(line.payload, new  
TokenSorter(TokenSorterType.INDEX_ASC));
```

File: `AssemblerThumb.java`

Method: `BuildBinOpCode`

Snippet:

```
1 Collections.sort(token.payload, new  
TokenSorter(TokenSorterType.INDEX_ASC));
```

An example of the `TokenSorter` class in use.

The first example shown in the previous listing is from the `AssemblerThumb` class' `BuildBinOpCode` method. This method is used to build the binary representation of an opcode and to do so it must make sure that the line's tokens are sorted by `index` so that the opcode's arguments are guaranteed in the correct order. To achieve this the code on line 1 of the example if used to sort the `TokenLine`'s payload, `line.payload`, by `index` ascending.

There are a few places where we have to concern ourselves with sorting arguments. One is for a non-group or list opcode, as we've just seen. The second example is for a group or list opcode. In this case we have to concern ourselves with sorting sub-tokens as the arguments for group or list instructions are kept in that token's `payload` field, not the `TokenLine`'s `payload` field.

Note that the call to sort the sub-tokens is almost identical to the previous example save for the location of the list of tokens to sort. In this case

we use the `token.payload` list because we're working with sub-tokens of a group or list. In the next section we'll take a look at a very important aspect of the `TOKENERS` package, the `Tokener` interface.

## Interface Review: Tokener.java

The `Tokener` interface is very similar to some of the interfaces we've looked at thus far in the base class review. Recall the purpose of the interface is to allow for different implementations. It provides a description of the methods necessary to implement in order to define a class that can plug into the assembler as a proper implementation of the `Tokener` interface. Let's jump into some code.

*Listing 11-23. Tokener.java Full Interface Review*

```
01 package net.middlemind.GenAsm.TOKENERS;  
02  
03 import net.middlemind.GenAsm.Lexers.ArtifactLine;  
04 import net.middlemind.GenAsm.Exceptions.Thumb.ExceptionNoTokenerFound;  
05 import net.middlemind.GenAsm.JsonObjs.JsonObj;  
06 import java.util.List;  
07  
08 public interface Tokener {  
09     public TokenLine LineTokenize(ArtifactLine line, int lineNumber,  
JsonObj entryTypes) throws ExceptionNoTokenerFound;  
10     public List<TokenLine> FileTokenize(List<ArtifactLine> file, jsonObj  
entryTypes) throws ExceptionNoTokenerFound;  
11 }
```

A full listing of the `Tokener` interface.

The `Tokener` interface, as noted earlier, defines the minimum set of methods that a class must implement in order to create a `Tokener` that can plug into the `GenAsm` assembler. Notice that there are two methods defined in the interface. The first method is designed to tokenize a single line and the second method is designed to tokenize an entire file. Let's take a moment to look over the method's arguments.

The first method shown, `LineTokenize`, takes three arguments. The first is the `ArtifactLine` instance, `line`. This argument provides us with the base information that will be converted into tokens. The second argument is the number of the line that we're working on, and the third argument is a set of entry types.

Recall from our review of the JSON data files that the `is_entry_types.json` file defines the different token types of the given instruction set. The second method listed, `FileTokenize`, takes two arguments. This first argument is a list of `ArtifactLine` instances, one for each line in the assembly source code file. The second argument for this method is the set of entry types for the given instruction set. In the next section we'll wrap up the chapter with a demonstration of the `Tokener` interface in use.

## Demonstration: Tokener.java

The demonstration of the `Tokener` interface contains a few examples of the interface in use including an instruction set specific implementation, `TokenerThumb`.

*Listing 11-24. Tokener.java Demonstration*

File: `TokenerThumb.java`

Method: `LineTokenize`

Snippet:

```
1 public TokenLine LineTokenize(ArtifactLine line, int lineNumber, JsonObject
entryTypes) throws ExceptionNoTokenerFound {}
```

File: `TokenerThumb.java`

Method: `FileTokenize`

Snippet:

```
1 public List<TokenLine> FileTokenize(List<ArtifactLine> file, JsonObject
entryTypes) throws ExceptionNoTokenerFound {}
```

File: `AssemblerThumb.java`

Method: `TokenizeLexerArtifacts`

Snippet:

```
1 TokenerThumb tok = new TokenerThumb();
2 asmDataTokened = tok.FileTokenize(asmDataLexed, jsonObjIsEntryTypes);
```

The first two examples, shown in the previous listing, are from the `TokenerThumb` class' implementation of the `Tokener` interface. Notice that the two methods defined in the interface are fully defined in the implementing class. The third example shown is an example of the `TokenerThumb` class in use. Take a moment to look over the short snippet of code associated with this example. Notice how easy it is to tokenize the lexed data. Simply initialize the `TokenerThumb` class, as on line 1, then call the `FileTokenize` method with the lexed data and loaded instruction set entry types as arguments. That's all it takes to tokenize the lexerized data.

## Chapter Conclusion

In this chapter we took a look at a lot of classes. Building on top of your experience with the ARM Thumb-1 instruction set, the preprocessor directives, the opcodes, the standard directives, and the JSON data files. We've now completed a full review of all the base classes and interfaces that support the Thumb-1 instruction set as implemented in the GenAsm assembler.

In the upcoming chapters we'll take a look at the Thumb-1 implementation in detail and cover all the steps, and classes, necessary to convert assembly source code into a binary program. Let's review the classes we've covered in this chapter.

- **Lexers Package**
  - **Artifact.java:** The `Artifact` class holds a sub-string of text from a line of assembly code.
  - **ArtifactLine.java:** The `ArtifactLine` class holds a list of `Artifact` object instances from one line of assembly source code.
  - **Lexer.java:** The `Lexer` interface defines the minimum requirements to implement a lexer that can plug into the GenAsm

assembler.

- **Linkers Package**
  - **Linker.java**: An interface that defines the minimum requirements necessary to create a `Linker` class that plugs into the GenAsm assembler.
- **Loaders Package (Loader Classes)**
  - **Loader.java**: An interface that defines how to create a basic JSON file loader class.
  - **LoaderBitSeries.java**: A loader class implementation that is designed to load a JSON file of bit series sub-object. This class is rarely used.
  - **LoaderIsSets.java**: A loader class implementation that is designed to load a JSON file of instruction set definition objects.
  - **LoaderLineHexReps.java**: A loader class implementation that is designed to load a JSON hex line representations file.
- **PreProcessors Package**
  - **PreProcessor.java**: An interface that defines the minimum set of requirements for creating a preprocessor that plugs into the GenAsm assembler.
- **Tokeners Package**
  - **Token.java**: A class that holds a `Token` object that is created from, extracted from, a given `Artifact` object instance.
  - **TokenLine.java**: A class that holds the `Tokens` extracted from a given `ArtifactLine` object instance.
  - **TokenSorter.java**: A class that provides a standard way to sort a list of `Token` objects.
  - **Tokener.java**: An interface that defines the minimum set of requirements to create a `Tokener` that can plug into the GenAsm assembler.

It can be difficult to visualize, or otherwise absorb this much information so don't be discouraged if you don't feel like you have it down completely. Take the time to review the chapter again if need be. I'm sure you'll find this chapter provides a solid reference and example of the project's base classes. In the upcoming chapters we'll be looking at the concrete implementation of an assembler using the structure and code provided by the base classes. As I'm sure you're aware of by now, we'll be using the ARM Thumb-1 instruction set as our example implementation.

# **Chapter 12: The Static Main Entry Point**

In the last two chapters we've covered a ton of material. Ironically, we haven't had much of a chance to use the formal class review process we outlined due to the concise nature of most of the base classes and interfaces.

In this chapter we're going to take things a bit further as we begin our exploration of the GenAsm assembler's key processes and Thumb-1 instruction set implementation. We'll build on the knowledge you've gained this far in the text, and we'll start with a detailed review of the starting point of the project. The static main entry point.

## **Class Review: GenAsm**

The `GenAsm` class, as mentioned earlier, is the class that holds the static main entry point to the GenAsm assembler program. As such it handles the command line arguments and starts off the assembly process. We'll review the class using the afore mentioned class review template. The relevant sections are as follows:

- Static Class Members

- Class Headers
- Demonstration

Because the `GenAsm` class is the static main entry point for our project it contains a fair number of static class members. We'll review them first then take a look at the class declaration and lastly, we'll demonstrate the class in action. Let's take a look at some code.

## Static Class Members: GenAsm

The `GenAsm` class has a number of static class fields that are used to configure the assembly process. Let's take a look.

*Listing 12-1. GenAsm.java Static Class Members 1*

```
public static String ASM_SETS_FILE_NAME = "";
public static String ASM_TARGET_SET = "";
public static String ASM_SETS_LOADER_CLASS = "";
public static String ASM_SETS_TARGET_CLASS = "";
public static JsonObjectIsSets ASM_SETS = null;

public static JsonObjectIsSet ASM_SET = null;
public static String ASM_ASSEMBLER_CLASS = "";
public static Assembler ASM_ASSEMBLER = null;
public static String ASM_ASSEMBLY_SOURCE_FILE = "";
public static String ASM_LINKER_CLASS = "";

public static Linker ASM_LINKER = null;
public static String ASM_PREPROCESSOR_CLASS = null;
public static PreProcessor ASM_PREPROCESSOR = null;
public static String ASM_ROOT_OUTPUT_DIR = "";

public static int ARG_LEN_TARGET = 12;
public static boolean ASM_VERBOSE = false;
public static boolean ASM_QUELL_FILE_OUTPUT = false;
public static String CFG_DIR_PATH = null;
```

A listing of the `GenAsm` class' static class fields.

The first static class field, `ASM_SETS_FILE_NAME`, is a string that references the path to the `is_sets.json` data file. This data file is the only data file that exists as part of the assembler's data and not as part of the instruction set. The next field, `ASM_TARGET_SET`, is a string that determines which instruction set to use to assemble the specified source code file.

The next two static class fields are, by design, part of the project's extensibility. The `ASM_SETS_LOADER_CLASS` and `ASM_SETS_TARGET_CLASS` fields are used to define the classes that should be loaded up and used to process the instruction set. The next two static class fields, `ASM_SETS` and `ASM_SET`, are used to hold the resulting data loaded from the `is_sets.json` data file.

Following these fields is the `ASM_ASSEMBLER_CLASS` which defines the class to load and use to process the assembly source file. Similarly, the static class field, `ASM_ASSEMBLER`, is used to hold the resulting instantiated class, recall that the target class must implement the `Assembler` interface, thus the field is an instance of the `Assembler` class. The `ASM_ASSEMBLY_SOURCE_FILE` field holds the path to the assembly source code to process.

The subsequent four static class fields follow a similar pattern to some of the fields we've reviewed thus far. The `ASM_LINKER_CLASS` field is used to hold the name of the class that will be loaded and used as the linker. The `ASM_LINKER` field holds a reference to the linker class loaded for this assembler run.

We can see a similar pattern when dealing with the preprocessor. The static class field, `ASM_PREPROCESSOR_CLASS`, is a string that references the full name of the class to load and use as the preprocessor. The `ASM_PREPROCESSOR` field holds a reference to the preprocessor class loaded for this assembler execution.

The next static class field, `ASM_ROOT_OUTPUT_DIR`, is a string that defines the location of all assembler output. You can use this field to change the location of the assembler's output files. The subsequent field, `ARG_LEN_TARGET`, defines an integer value that indicates the number of program arguments that must be defined for the `GenAsm` class to switch from

using hard coded configuration values to data driven command line arguments.

While this may seem superfluous it really comes in handy during development because you can hard code the `GenAsm` class fields to work with your current assembly program. This allows you to easily run the program, properly configured, through the NetBeans IDE. The following two static fields, `ASM_VERBOSE` and `ASM_QUELL_FILE_OUTPUT`, are Boolean flags that are used to turn off verbose logging and file output respectively.

These fields can be used to optimize the assembler a little bit by preventing unnecessary logging and output from being generated. The last static field in the list, `CFG_DIR_PATH`, is used to set the path to the configuration directory used by the assembler. In the next section we'll take a look at the `GenAsm` class' only method, the static main method. The method is somewhat long so I've broken it up into two parts. I recommend taking a moment to quickly review the entire method before continuing along here.

*Listing 12-2. GenAsm.java Static Class Members 2*

```
01 public static void main(String[] args) throws Exception {
02     if(args != null && args.length == 3) {
03         String cmd = args[0];
04         if (Utils.IsEmpty(cmd) == false && cmd.equals("compare") ==
true) {
05             String f1 = args[1];
06             String f2 = args[2];
07             byte[] b1 = FileLoader.LoadBin(f1);
08             byte[] b2 = FileLoader.LoadBin(f2);
09             Logger.wrl("File length 1: " + b1.length);
10             Logger.wrl("File length 2: " + b2.length);
11             if (b1.length == b2.length) {
12                 for (int i = 0; i < b1.length; i++) {
13                     String h1 = Integer.toHexString(b1[i]);
14                     String h2 = Integer.toHexString(b2[i]);
15                     if (b1[i] != b2[i]) {
16                         Logger.wrl(i + " " + (i / 2) + " h1: " +
Utils.FormatHexString(h1, 2, true) + " h2: " + Utils.FormatHexString(h2,
2, true) + " DIFF");
17                     } else {
```

```

18                         Logger.wrl(i + " " + (i / 2) + " h1: " +
Utils.FormatHexString(h1, 2, true) + " h2: " + Utils.FormatHexString(h2,
2, true));
19                     }
20                 }
21             }
22         return;
23     }
24 } else if(args == null || args.length < ARG_LEN_TARGET) {
25     String targetProgram = "TEST_N_AsmChecks";
26     CFG_DIR_PATH = "C:\\\\Users\\\\variable\\\\Documents\\\\GitHub\\\\GenAsm\\\\
cfg\\\\";
27     ASM_SETS_FILE_NAME = "./cfg/is_sets.json";
28     ASM_TARGET_SET = "THUMB_ARM7TDMI";
29     ASM_SETS_LOADER_CLASS =
"net.middlemind.GenAsm.Loaders.LoaderIsSets";
30     ASM_SETS_TARGET_CLASS =
"net.middlemind.GenAsm.JsonObjs.JsonObjIsSets";
31     ASM_SETS = null;
32     ASM_SET = null;
33     ASM_ASSEMBLER_CLASS =
"net.middlemind.GenAsm.Assemblers.ThumbAssemblerThumb";
34     ASM_ASSEMBLER = null;
35     ASM_ASSEMBLY_SOURCE_FILE = CFG_DIR_PATH + "THUMB\\\\TESTS\\\\" +
targetProgram + "\\genasm_source.txt";
36     ASM_LINKER_CLASS =
"net.middlemind.GenAsm.Linkers.ThumbLinkerThumb";
37     ASM_LINKER = null;
38     ASM_PREPROCESSOR_CLASS =
"net.middlemind.GenAsm.PreProcessors.ThumbPreProcessorThumb";
39     ASM_PREPROCESSOR = null;
40     ASM_ROOT_OUTPUT_DIR = CFG_DIR_PATH + "THUMB\\\\OUTPUT\\\\" +
targetProgram + "\\";
41     ASM_VERBOSE = false;
42     ASM_QUELL_FILE_OUTPUT = false;
43 } else if(args != null || args.length == ARG_LEN_TARGET) {
44     CFG_DIR_PATH = args[11];
45     ASM_SETS_FILE_NAME = args[0];
46     ASM_TARGET_SET = args[1];
47     ASM_SETS_LOADER_CLASS = args[2];
48     ASM_SETS_TARGET_CLASS = args[3];
49     ASM_SETS = null;
50     ASM_SET = null;

```

```

51     ASM_ASSEMBLER_CLASS = args[4];
52     ASM_ASSEMBLER = null;
53     ASM_ASSEMBLY_SOURCE_FILE = args[5];
54     ASM_LINKER_CLASS = args[6];
55     ASM_LINKER = null;
56     ASM_PREPROCESSOR_CLASS = args[7];
57     ASM_PREPROCESSOR = null;
58     ASM_ROOT_OUTPUT_DIR = args[8];
59     ASM_VERBOSE = Boolean.parseBoolean(args[9]);
60     ASM_QUELL_FILE_OUTPUT = Boolean.parseBoolean(args[10]);
61 } else {
62     System.out.println("Error: incorrect number of arguments.");
63     System.out.println("Please provide the following arguments in the
order shown, examples included.");
64     System.out.println("1 \"./cfg/is_sets.json\" (ASM_SETS_FILE_NAME)");
65     System.out.println("2 \"THUMB_ARM7TDMI\" (ASM_TARGET_SET)");
66     System.out.println("3 \"net.middlemind.GenAsm.Loaders.LoaderIsSets\""
(ASM_SETS_LOADER_CLASS));
67
System.out.println("4 \"net.middlemind.GenAsm.JsonObjs.JsonObjIsSets\""
(ASM_SETS_TARGET_CLASS));
68
System.out.println("5 \"net.middlemind.GenAsm.Assemblers.ThumbAssemblerTh
umb\" (ASM_ASSEMBLER_CLASS)");
69     System.out.println("6 \"C:\\\\Users\\\\variable\\\\Documents\\\\GitHub\\\\
GenAsm\\\\cfg\\\\THUMB\\\\TESTS\\\\TEST_N_AsmChecks\\\\genasm_source.txt\""
(ASM_ASSEMBLY_SOURCE_FILE));
70
System.out.println("7 \"net.middlemind.GenAsm.Linkers.ThumbLinkerThumb\""
(ASM_LINKER_CLASS));
71
System.out.println("8 \"net.middlemind.GenAsm.PreProcessors.Thumb.PreProce
ssorThumb\" (ASM_PREPROCESSOR_CLASS)");
72     System.out.println("9 \"C:\\\\Users\\\\variable\\\\Documents\\\\GitHub\\\\
GenAsm\\\\cfg\\\\THUMB\\\\OUTPUT\\\\TEST_N_AsmChecks\\\\\" (ASM_ROOT_OUTPUT_DIR)");
73     System.out.println("10 false (ASM_VERBOSE)");
74     System.out.println("11 false (ASM_QUELL_FILE_OUTPUT)");
75     System.out.println("12 \"C:\\\\Users\\\\variable\\\\Documents\\\\GitHub\\\\
GenAsm\\\\cfg\\\\\" (CFG_DIR_PATH)");
76 }

```

The first part of the GenAsm class' main method listing, lines 1 – 76.

The GenAsm class' main method is a bit lengthy so we'll separate it into two parts. In the previous listing the first part of the method is shown. There are three distinct cases in the code snippet shown. The first case, on lines 2 – 23, is for utility, as you'll soon see. The second case, on lines 24 – 42, is meant to handle a default hard coded case, the program you're working on. The third case, found on lines 43 – 60, is the data driven case. This section handles the command line arguments passed to the assembler.

These arguments are represented in the GenAsm class' fields we've just reviewed. Proceeding with a detailed review of the method, on line 2, if the args argument is defined and has only three entries then we continue to the utility case. This type of command line argument uses the following pattern:

```
"compare"
```

```
"C:\\\\Users\\\\variable\\\\Documents\\\\GitHub\\\\GbaAssemblyChecks\\\\
chibi_akumas\\\\ARMDevTools\\\\gba_example4_hello_world_thumb\\\\
program.gba"

"C:\\\\Users\\\\variable\\\\Documents\\\\GitHub\\\\GbaAssemblyChecks\\\\
chibi_akumas\\\\ARMDevTools\\\\gba_example4_hello_world_thumb\\\\
output_assembly_listing_endian_lil.bin"
```

You can inject the arguments with the following line of code if you don't want to wrangle with command line arguments.

```
args = new String[] {
    "compare",
    "C:\\\\Users\\\\variable\\\\Documents\\\\GitHub\\\\
GbaAssemblyChecks\\\\chibi_akumas\\\\ARMDevTools\\\\
gba_example4_hello_world_thumb\\\\program.gba",
```

```
"C:\\Users\\variable\\Documents\\GitHub\\
GbaAssemblyChecks\\chibi_akumas\\ARMDevTools\\
gba_example4_hello_world_thumb\\
output_assembly_listing_endian_lil.bin"
};

};
```

The previous examples show files that only exist on my computer so your paths will differ. This tool compares the two binary files and indicates which bytes are different. This comes in handy when comparing say the output of an assembler project against a known assembler's output.

On line 24, if the `args` argument is defined and the number of entries matches the `ARG_LEN_TARGET` class field. In this case, lines 25 – 42, each key class field is hard coded with a value that is pertinent to the program you are working on. In this case the settings are those from my development computer, and the “TEST\_N\_AsmChecks” program in particular.

Take a moment to look over the settings. You'll notice that they are configured for the Thumb-1 instruction set and the specific classes to be used are specified as part of the configuration. In this way the program is very extensible. You can provide your own classes as long as they implement the requisite interface. The last case, lines 44 – 60, is the general case where all the class fields are driven from the command line arguments. Last but not least, on lines 62 – 75, the error message is printed showing the order, meaning, and an example of the given argument.

### *Listing 12-3. GenAsm.java Static Class Members 3*

```
01 if(Utils.IsEmpty(ASM_SETS_FILE_NAME)) {
02     Logger.wrlErr("GenAsm: Main: Error: No assembly source file
provided.");
03 } else if(Utils.IsEmpty(ASM_TARGET_SET)) {
04     Logger.wrlErr("GenAsm: Main: Error: No assembly target set
provided.");
05 } else if(Utils.IsEmpty(ASM_SETS_LOADER_CLASS)) {
06     Logger.wrlErr("GenAsm: Main: Error: No assembly set loader
provided.");
07 } else {
```

```

08     String json;
09     Class cTmp;
10     LoaderIsSets ldrIsSets;
11
12     try {
13         json = FileLoader.LoadStr(ASM_SETS_FILE_NAME);
14     } catch (IOException e) {
15         Logger.wrl("GenAsm: Main: Error: Could not load is_sets file " +
ASM_SETS_FILE_NAME);
16         e.printStackTrace();
17         return;
18     }
19
20     if (!Utils.IsEmptyString(json)) {
21         try {
22             cTmp = Class.forName(ASM_SETS_LOADER_CLASS);
23             ldrIsSets =
(LoaderIsSets)cTmp.getDeclaredConstructor().newInstance();
24             ASM_SETS = ldrIsSets.ParseJson(json, ASM_SETS_TARGET_CLASS,
ASM_SETS_FILE_NAME);
25             cTmp = Class.forName(ASM_PREPROCESSOR_CLASS);
26             ASM_PREPROCESSOR =
(PreProcessor)cTmp.getDeclaredConstructor().newInstance();
27             cTmp = Class.forName(ASM_ASSEMBLER_CLASS);
28             ASM_ASSEMBLER =
(Assembler)cTmp.getDeclaredConstructor().newInstance();
29             cTmp = Class.forName(ASM_LINKER_CLASS);
30             ASM_LINKER =
(Linker)cTmp.getDeclaredConstructor().newInstance();
31
32             for (JsonObjIsSet entry : ASM_SETS.is_sets) {
33                 if (entry.set_name.equals(ASM_TARGET_SET)) {
34                     Logger.wrl("GenAsm: Main: Found instruction set entry " +
ASM_TARGET_SET);
35                     ASM_SET = entry;
36                     break;
37                 }
38             }
39
40             if (ASM_SET != null) {
41                 if (ASM_ASSEMBLER != null && ASM_PREPROCESSOR != null &&
ASM_LINKER != null) {

```

```

42             List<String> fileData =
ASM_PREPROCESSOR.RunPreProcessor(ASM_ASSEMBLY_SOURCE_FILE,
ASM_ROOT_OUTPUT_DIR, null);
43             ASM_ASSEMBLER.RunAssembler(ASM_SET,
ASM_ASSEMBLY_SOURCE_FILE, fileData, ASM_ROOT_OUTPUT_DIR, null, null,
ASM_VERBOSE, ASM_QUELL_FILE_OUTPUT);
44             ASM_LINKER.RunLinker(ASM_ASSEMBLER,
ASM_ASSEMBLY_SOURCE_FILE, ASM_ROOT_OUTPUT_DIR, null, ASM_VERBOSE,
ASM_QUELL_FILE_OUTPUT);
45         } else {
46             Logger.wrl("GenAsm: Main: Error: could not find properly
loaded pre-processor, assembler, or linked");
47         }
48     } else {
49         Logger.wrl("GenAsm: Main: Error: could not find assembler
set named " + ASM_TARGET_SET);
50     }
51 } catch (ExceptionLoader | ClassNotFoundException |
InstantiationException | IllegalAccessException | NoSuchMethodException |
InvocationTargetException e) {
52     Logger.wrl("GenAsm: Main: Error: could not instantiate loader
class " + ASM_SETS_LOADER_CLASS);
53     e.printStackTrace();
54 }
55 }
56 }
```

The second part of the GenAsm class' main method listing.

The remainder of the `main` method is shown in the previous listing. On lines 1 – 7 we put the `Utils` class to use and validate some important class fields. On lines 8 – 10 a few method variables are declared and on lines 12 – 18 the `is_sets.json` data files is loaded into the local string, `json`. On line 20 we check if the file contains data.

Next, on line 22 we create an instance of the loader class specified to load the `is_sets.json` data file. This class has to extend the `LoaderIsSets` class because a new instance of the class is set on line 23. The code casts the specified class to the `LoaderIsSets` superclass and stores the

result in the `ldrIsSets` method variable which is an instance of the `LoaderIsSets` class.

In this way you can introduce new code into the `GenAsm` assembler by defining your own loader class that extends the `LoaderIsSets` class. On line 24 the JSON data is loaded into the `ASM_SETS` static class field which is an instance of the `JsonObjIsSets` class. Recall from prior material that this is an instance of a holder class. Take a moment to think about that. In just these few lines of code we've seen JSON data files, a loader class, and a holder class all working together to load data into the assembler in an extensible way.

The code on lines 25 – 30 follows the same pattern as the code we've just reviewed but without the data load. In each case we are creating a new instance of a class, specified by a command line argument, and casting it to its super class for use in the assembly process. Throughout the code you'll notice certain error handling. I'm probably not going to address this directly since we have so much other code to cover but keep an eye out for the use of exceptions and error handling in the code.

On lines 32 – 38 we iterate over the list of loaded instruction sets and attempt to find the set specified by the `ASM_TARGET_SET` field. If we're not able to locate the desired instruction set errors are reported and the program exits. Otherwise, we proceed and start the assembly process. I'll reiterate the process that the `GenAsm` assembler employs here.

1. Enter static main, process arguments.
2. Load specified assembly source code file.
3. Run preprocessor directives, update the assembly source code accordingly.
4. Run the Lexer and break the assembly source code down into pieces of text.
5. Run the tokenizer and scan each artifact to identify the type of text it contains, and the type of token that will be used to represent it, using the text match rules defined in the `is_entry_types.json` data file.
6. Run the assembler and convert the list of token lines to binary data.
7. Run the linker and finalize the binary program's structure before writing the final output file.

At this point in the class review we've just completed step 1 and we're about to start on step 2. Let's finish up our review of the `main` method. If the instruction set is found and the preprocessor, assembler, and linker are defined, lines 40 and 41, then we proceed to run the preprocessor on line 42. Followed by the assembler, line 43, and the linker on line 44.

You may be wondering where the lexer and tokener are. Well because the lexer and tokener can depend heavily on the instruction set you're trying to implement I left them to be run as part of the `Assembler` class' responsibilities. One could make the argument that the linker should also be part of the assembler. You might be right but in this implementation the linker is run by the `GenAsm` class and not the `Assembler`. That brings us to the conclusion of the `GenAsm` static class members review. In the next section we'll take a quick look at the class' declaration.

## Class Headers: GenAsm

The class declaration for the `GenAsm` class is as follows. Take a moment to look over the imports used by the class.

*Listing 12-4. GenAsm.java Class Headers*

```
package net.middlemind.GenAsm;

import net.middlemind.GenAsm.FileIO.FileLoader;
import net.middlemind.GenAsm.AssemblersAssembler;
import net.middlemind.GenAsm.Exceptions.Thumb.ExceptionLoader;
import net.middlemind.GenAsm.JsonObjs.JsonObjIsSet;
import net.middlemind.GenAsm.JsonObjs.JsonObjIsSets;
import net.middlemind.GenAsm.Loaders.LoaderIsSets;
import java.io.IOException;
import java.lang.reflect.InvocationTargetException;
import java.util.List;
import net.middlemind.GenAsm.Linkers.Linker;
import net.middlemind.GenAsm.PreProcessors.PreProcessor;

public class GenAsm {}
```

The `GenAsm` class declaration.

The class declaration is simple and direct. There's not much to discuss here so we'll move on to the demonstration of the `GenAsm` class.

## Demonstration: GenAsm

To demonstrate the `GenAsm` class we'll take a look at an example of calling the program from the command line. We'll be sure to discuss the directory structure and any other requirements as we go. In general, you should make sure that the directories you specify exist. For our example we'll pretend we've installed the `GenAsm` assembler on our computer in a given folder. In our example setup we have the `GenAsm` jar and the configuration directory in the same parent folder. The `is_sets.json` data file is located at the root of the configuration folder. I'll detail the directory structure here.

*Listing 12-5. GenAsm Example Program Directory Structure*

```
-> Assembler Install Directory
    -> GenAsm.jar
    -> cfg
        -> is_sets.json
    -> THUMB
        -> <instruction set data files here>
        -> OUTPUT
            -> TEST_N_AsmChecks
    -> TESTS
        -> TEST_N_AsmChecks
        -> genasm_source.txt
```

An example of a `GenAsm` execution environment.

Our intention is to use the program from the command line, so we'll cover running the program directly here. I've used the paths that are local to my machine but, it can still give you an idea of how you might call the program. Because it gets a bit tedious to read through a list of the arguments on one line, as they would be in a terminal, I've separated them out by line here for convenience.

### *Listing 12-6. GenAsm Example Program Execution*

```
java -jar GenAsm.jar
01 "./cfg/is_sets.json"
02 "THUMB_ARM7TDMI"
03 "net.middlemind.GenAsm.Loaders.LoaderIsSets"
04 "net.middlemind.GenAsm.JsonObjs.JsonObjIsSets"
05 "net.middlemind.GenAsm.Assemblers.ThumbAssemblerThumb"
06 "C:\Users\variable\Documents\GitHub\GenAsm\cfg\THUMB\TESTS\
TEST_N_AsmChecks\genasm_source.txt"

07 "net.middlemind.GenAsm.Linkers.ThumbLinkerThumb"
08 "net.middlemind.GenAsm.PreProcessors.ThumbPreProcessorThumb"
09 "C:\Users\variable\Documents\GitHub\GenAsm\cfg\THUMB\OUTPUT\
TEST_N_AsmChecks\"

10 "false"
11 "false"
12 "C:\Users\variable\Documents\GitHub\GenAsm\cfg\"
```

An example of calling the GenAsm program with arguments separated by line breaks for clarity. Each argument line starts with the index of the argument for clarity. Do not include the argument indexes in actual use.

That brings us to the conclusion of this section. Make sure you are familiar with the execution of the software and can competently setup an execution directory outside of the NetBeans project folder if you intend to use the software further. If not, it should suffice to use the hard coded section of the static main entry point so that you can execute the example programs quickly and easily, with no arguments needed.

## **Chapter Conclusion**

And with that we've reached the conclusion of this chapter. We're progressing slowly, adding more to the knowledge that we've built up thus far. We've started peeking under the hood taking a look at how the assembler works. Let's summarize what we've covered in this chapter.

- **Arguments:** We took a look at the different arguments that the GenAsm program accepts. We looked at the utility calling arguments, where two binary files are compared and differences listed, as well as a full listing of command line arguments.
- **Execution Order:** In this chapter we took another look at the execution order, the main steps of the assembler, and got an idea of what the assembler is doing underneath the hood.
- **Program Directory Structure:** An example of the directory structure, environment, necessary to run the GenAsm assembler.
- **Example Program Execution:** We took a look at how to call the program from the command line with a full set of arguments.

Take a moment to think about the code we've just reviewed in the context of the greater process. Also, think about the instruction set you've reviewed and how the base classes – loaders, holders, and helpers – work together to create an extensible environment to build out an instruction set and start assembling some programs. We've covered a lot of classes thus far and we have a few more to go. In the next chapter we'll slow things down a little and take a look at the instruction set specific – loader, holder, and helper – classes.

# **Chapter 13: The Holder, Loader, and Exception Classes**

We've covered a lot of heavy material up to this point in the text. This chapter is going to be a light one. I want to spend a little time reviewing the ARM Thumb-1 instruction set's loader, holder, and exception classes. We've seen each of these during our review of the project's base classes. This time we'll look at them from the perspective of an instruction set implementation.

There are a number of classes implemented for each type – holder, loader, exception. Due to the redundancy of the classes and their implementation we'll only be looking at a few example classes in each category. In any case, make sure to review the classes we don't cover here on your own. I will make sure to list the classes that are omitted from direct review.

## **Thumb-1 Holder Classes**

The ARM Thumb-1 instruction set implementation has a number of holder classes that are specific to this instruction set. As such they exist in packages that indicate the instruction set they belong to. Also, the naming convention is

such that the base package that the instruction set implementation is extending, for lack of a better term, is also the literal base package of the class.

For instance, the base holder package is the `net.middlemind.GenAsm.JsonObjs` package and the Thumb-1 specific implementation belongs to the `net.middlemind.GenAsm.JsonObjs.Thumb` package. This naming convention makes clear the relationship between the instruction set implementation and the base classes they are based on. It also clearly indicates what classes belong to which instruction sets. Let's take a moment to list the holder classes of the Thumb-1 implementation.

*Listing 13-1. ARM Thumb-1 Specific Holder Classes*

Class	Type	JSON File (If Applicable)
<code>JsonObjIsArgType.java</code>	Object	<code>is_arg_types.json</code>
<code>JsonObjIsArgTypes.java</code>	List	<code>is_arg_types.json</code>
<code>JsonObjIsDirective.java</code>	Object	<code>is_directives.json</code>
<code>JsonObjIsDirectiveArg.java</code>	Sub-Object	<code>is_directives.json</code>
<code>JsonObjIsDirectives.java</code>	List	<code>is_directives.json</code>
<code>JsonObjIsEmptyDataLines.java</code>	List	<code>is_empty_data_lines.json</code>
<code>JsonObjIsEntryType.java</code>	Object	<code>is_entry_types.json</code>
<code>JsonObjIsEntryTypes.java</code>	List	<code>is_entry_types.json</code>
<code>JsonObjIsOpCode.java</code>	Object	<code>is_op_codes.json</code>
<code>JsonObjIsOpCodeArg.java</code>	Sub-Object	<code>is_op_codes.json</code>
<code>JsonObjIsOpCodeArgSorter.java</code>	Helper	-
<code>JsonObjIsOpCodes.java</code>	List	<code>is_op_codes.json</code>
<code>JsonObjIsRegister.java</code>	Object	<code>is_registers.json</code>
<code>JsonObjIsRegisters.java</code>	List	<code>is_registers.json</code>
<code>JsonObjIsTestFileVal.java</code>	Object	-
<code>JsonObjIsTestFileVals.java</code>	List	-
<code>JsonObjIsValidLine.java</code>	Object	<code>is_valid_lines.json</code>
<code>JsonObjIsValidLineEntry.java</code>	Sub-Object	<code>is_valid_lines.json</code>
<code>JsonObjIsValidLines.java</code>	List	<code>is_valid_lines.json</code>

A list of the holder, `JsonObj`, classes used to implement the ARM Thumb-1 instruction set.

There are a lot of holder classes shown in the previous listing. We won't be reviewing all of them, but I do want to take a look at a few specific classes. The classes that we'll be reviewing in this section are as follows.

- **JsonObjIsArgTypes.java**: Demonstrates a list holder class.
- **JsonObjIsRegister.java**: Demonstrates an object holder class.
- **JsonObjIsValidLineEntry.java**: Demonstrates a sub-object holder class.

As you review the classes in this section keep in mind your experience with the project's base holder classes we reviewed earlier in the text. Let's take a look at some code.

## Class Review: **JsonObjIsArgTypes.java**

The `JsonObjIsArgTypes` class is used to hold data loaded from the `is_arg_types.json` data file. This class exemplifies the list type of holder class. Due to the concise nature of the class, we'll forego the longer review template and just list the class in its entirety.

*Listing 13-2. JsonObjIsArgTypes.java Full Class Listing*

```
01 package net.middlemind.GenAsm.JsonObjs.Thumb;
02
03 import java.util.ArrayList;
04 import java.util.List;
05 import net.middlemind.GenAsm.Exceptions.Thumb.ExceptionJsonObjLink;
06 import net.middlemind.GenAsm.JsonObjs.JsonObj;
07 import net.middlemind.GenAsm.JsonObjs.JsonObjBase;
08 import net.middlemind.GenAsm.Logger;
09 import net.middlemind.GenAsm.Utils;
10
11 public class JsonObjIsArgTypes extends JsonObjBase {
12     public String obj_name;
13     public String set_name;
```

```

14     public List<JsonObjIsArgType> is_arg_types;
15
16     @Override
17     public void Link(JsonObj linkData) throws ExceptionJsonObjLink {
18         for(JsonObjIsArgType entry : is_arg_types) {
19             entry.linked_is_entry_types = new
ArrayList<JsonObjIsEntryType>();
20             for(String s : entry.is_entry_types) {
21                 boolean found = false;
22                 for(JsonObjIsEntryType lentry :
((JsonObjIsEntryTypes)linkData).is_entry_types) {
23                     if(!Utils.IsEmpty(lentry.type_name) &&
lentry.type_name.equals(s)) {
24                         entry.linked_is_entry_types.add(lentry);
25                         found = true;
26                         break;
27                     }
28                 }
29             }
30             if(!found) {
31                 throw new ExceptionJsonObjLink("JsonObjIsArgTypes: Link:
Error: Could not find JsonObjIsEntryType, group or single, object with
name " + s);
32             }
33         }
34     }
35 }
36
37     @Override
38     public void Print() {
39         Print("");
40     }
41
42     @Override
43     public void Print(String prefix) {
44         super.Print(prefix);
45         Logger.wrl(prefix + "ObjectName: " + obj_name);
46         Logger.wrl(prefix + "SetName: " + set_name);
47
48         Logger.wrl(prefix + "IsArgTypes:");
49         for(JsonObjIsArgType entry : is_arg_types) {
50             Logger.wrl("");
51             entry.Print(prefix + "\t");

```

```
52     }
53 }
54 }
```

A complete listing of the `JsonObjIsArgTypes` class.

The first thing to notice is that on line 11 the `JsonObjIsArgTypes` class extends the `JsonObjBase` class which is a simple implementation of the `JsobObj` interface. This class has three fields for us to look at. The first field, `obj_name`, holds the proper name of the JSON object that this class is designed to hold. The `set_name` field is used to hold the name of the instruction set that this class belongs to. This information is carried over from the JSON data file because it is also associated with an instruction set so we don't need to rely on default values for the class fields.

The last field is what makes this holder class a list type holder class. The `is_arg_types` field holds a list of objects loaded from the JSON data file. This pattern of list, object, sub-object, and sometimes sub-sub-objects, is repeated over and over again for different data files in the Thumb-1 instruction set implementation.

Next, take a moment to look over the class methods. They are overridden methods that provide customized functionality that differs from the `JsonObjBase` class' implementation. Notice that the `Link` method is used to create class references between key objects where the JSON data only holds a name or other identifying field. In this case we're iterating over each `JsonObjIsArgType` object in the `is_arg_types` field to link its entry type data to actual entry type objects, lines 20 – 33. The print methods are overridden to display information pertinent to this class.

The next class we'll look at is the `JsonObjIsRegister` class. This is a holder class that's an example of the object type of class. These are main objects that are contained by a list type holder class. In almost all cases the list version of the class is plural, and the main object is singular. In the next section we'll take a look at how the class is used.

## Demonstration: JsonObjIsArgTypes.java

Because the `JsonObjIsArgTypes` class is a list type holder class we'll look at an example of how it's used by the `LoaderIsArgTypes` class as part of the class' data load.

*Listing 13-3. JsonObjIsArgTypes.java Demonstration*

File: `LoaderIsArgTypes.java`

Method: `ParseJson`

Snippet:

```
01 JsonObjIsArgTypes jsonObj =
(JsonObjIsArgTypes)Class.forName(targetClass).getConstructor().newInstance();
02 jsonObj = gson.fromJson(json, jsonObj.getClass());
03 jsonObj.name = targetClass;
04 jsonObj.fileName = fileName;
05 jsonObj.loader = getClass().getName();
06
07 for(JsonObjIsArgType entry : jsonObj.is_arg_types) {
08     entry.name = entry.getClass().getName();
09     entry.fileName = fileName;
10     entry.loader = getClass().getName();
11 }
```

A demonstration showing the `LoaderIsArgTypes` class creating, and populating, a new instance of the `JsonObjIsArgTypes` class.

On line 1 a new instance of the `JsonObjIsArgTypes` class is created using reflection and the provided full class name, `targetClass`. On line 2 the loader class is populated with a call to the `Gson` class' `fromJson` method. Note that this method takes a JSON string, `json`, and the class type as arguments. Once this call has completed the entire JSON object structure has been loaded including sub-objects. Remember loading classes work with JSON files, not JSON objects.

The remaining of the class' fields are populated on lines 3 – 5. Recall that this holder class is of the list type. As such it has a list of associated objects. On lines 7 – 11 we iterate over the sub-object entries and set some of

their class field values, line 8 – 10. In the next section we'll take a look at a holder class of the object type.

## Class Review: JsonObjIsRegister.java

The next class for us to look at is the `JsonObjIsRegister` class, listed subsequently, exemplifies an object type holder class. Expect this class to closely follow the implementation conventions found in holder classes we've reviewed thus far.

*Listing 13-4. JsonObjIsRegister.java Full Class Listing*

```
01 package net.middlemind.GenAsm.JsonObjs.Thumb;
02
03 import net.middlemind.GenAsm.JsonObjs.JsonObjBase;
04 import net.middlemind.GenAsm.JsonObjs.JsonObjBitRep;
05 import net.middlemind.GenAsm.Logger;
06
07 public class JsonObjIsRegister extends JsonObjBase {
08     public String obj_name;
09     public String register_name;
10     public String is_entry_type;
11     public JsonObjIsEntryType linked_is_entry_type;
12     public JsonObjBitRep bit_rep;
13     public String desc;
14
15     @Override
16     public void Print() {
17         Print("");
18     }
19
20     @Override
21     public void Print(String prefix) {
22         super.Print(prefix);
23         Logger.wrl(prefix + "ObjName: " + obj_name);
24         Logger.wrl(prefix + "RegisterName: " + register_name);
25         Logger.wrl(prefix + "Description: " + desc);
26
27         Logger.wrl(prefix + "BitRep:");
28         bit_rep.Print(prefix + "\t");
29 }
```

A complete listing of the `JsonObjIsRegister` class.

First, take a moment to look at line 7, notice that this class also extends the `JsonObjBase` class. This is by design. We're coding to the interface here. Each holder class has shared structure due to their implementation as extensions of a base class that implements a holder class interface, `JsonObj`. As you might have expected the first field in the class is the `obj_name` field. This field provides a proper name for the class that's driven by the JSON data file it's loaded from. The name field, `register_name`, contains the name of the register as it would appear in an assembly source code file, R0, R1, R2, etc.

The next field for us to review is the `is_entry_type` field and it's used to tie this object to a specific entry type. Entry types are the unique tokens that can exist in an assembly source code file for the Thumb-1 instruction set. This field holds the name of the entry type it's associated with while the `linked_is_entry_type` field holds a reference to the associated entry type object. This is also a common implementation pattern throughout the holder classes.

The next class field, `bit_rep`, is an example of a sub-object. These objects are usually smaller and belong to another sub-object or a main object. In this case the `bit_rep` field holds the binary representation of this register. In the next section we'll take a look at how this class is used.

## Demonstration: `JsonObjIsRegister.java`

The demonstration for the `JsonObjIsRegister` class is quite simple. In the subsequent example we'll look at how the `AssemblerThumb` class uses the list of known registers to identify a token as being of the type, opcode argument. Let's take a look.

*Listing 13-5. `JsonObjIsRegister.java` Demonstration*

File: `AssemblerThumb.java`

Method: `PopulateOpCodeAndArgData`

Snippet:

```
01 if (Utils.ArrayContainsString(JsonObjIsEntryTypes.NAME_REGISTERS,
token.type_name)) {
02     String regCode = token.source;
03     if (token.type_name.equals(JsonObjIsEntryTypes.NAME_REGISTERWB)) {
04         regCode = regCode.replace("!", "");
05     }
06     regCode = regCode.replace(" ", "");
07     regCode = regCode.replace(",", "");
08
09     for (JsonObjIsRegister register : jsonObjIsRegisters.is_registers) {
10         if (register.register_name.equals(regCode)) {
11             token.register = register;
12             token.isOpCodeArg = true;
13             break;
14         }
15     }
16 }
```

A demonstration showing the `JsonObjIsRegister` class in use.

The previously listed example shows the `JsonObjIsRegister` class in use by the `AssemblerThumb` class' `PopulateOpCodeAndArgData` method. On line 1 we use the `Utils` class' helper method, `ArrayContainsString`, to check if the token is of the type, register. Next, we initialize a local variable, `regCode`, to the source code of the token. This will be the syntactically correct register name.

On lines 3 – 5 we check to see if the register is a write back register, includes a ‘!’ character prefix, and clean the indicator prefix if need be. Otherwise, we iterate over the list of known registers and attempt to find a match with the `regCode` variable's value. If a match is found we record the register match, line 11, and mark the token as an opcode argument.

This will have more significance in latter chapters when we get into the core details of the assembly process. That concludes this simple example. The main takeaway here is that the JSON data loaded into the Thumb-1 assembler is used to drive how the program functions. What constitutes a register is defined by a few entries in a few JSON data files. The next class

we'll look into is the `JsonObjIsValidLineEntry` class. This class exemplifies a sub-object type holder class.

## Class Review: `JsonObjIsValidLineEntry.java`

The `JsonObjIsValidLineEntry` class is an example of a sub-object holder class. Note that this class follows the pattern we've seen time and time again with this type of class.

*Listing 13-6. `JsonObjIsValidLineEntry.java` Full Class Listing*

```
01 package net.middlemind.GenAsm.JsonObjs.Thumb;
02
03 import java.util.List;
04 import net.middlemind.GenAsm.JsonObjs.JsonObjBase;
05 import net.middlemind.GenAsm.Logger;
06
07 public class JsonObjIsValidLineEntry extends JsonObjBase {
08     public String obj_name;
09     public List<String> is_entry_types;
10     public List<JsonObjIsEntryType> linked_is_entry_types;
11     public int index;
12
13     @Override
14     public void Print() {
15         Print("");
16     }
17
18     @Override
19     public void Print(String prefix) {
20         super.Print(prefix);
21         Logger.wrl(prefix + "ObjName: " + obj_name);
22         Logger.wrl(prefix + "Index: " + index);
23
24         Logger.wrl(prefix + "IsEntryTypes:");
25         for (String s : is_entry_types) {
26             Logger.wrl(prefix + "\t" + s);
27         }
28
29         if (linked_is_entry_types != null) {
30             Logger.wrl(prefix + "IsEntryTypes:");


```

```
31     for (JsonObjIsEntryType entry : linked_is_entry_types) {  
32         Logger.wrl("");  
33         entry.Print(prefix + "\t");  
34     }  
35 }  
36 }  
37 }
```

A complete listing of the `JsonObjIsValidLineEntry` class.

The `JsonObjIsValidLineEntry` is very simple in nature. It's responsible for listing all the valid entry types that can occur on a line and in what order. When we talk about entry types, we really mean `Token` objects that have been defined as a certain entry type. We'll spend more time on this when we cover the `TokenerThumb` class a little later on in the text.

The first field of the class, line 8, is the `obj_name` field and it's used to hold the proper name of the object as defined by the JSON data file. The next field `is_entry_types`, is a string-based list that is also driven from the data file. The next field is not driven from the data file, the `linked_is_entry_types` field is populated during the linking phase of the JSON data file load. The last field in the class, `index`, is used to track the position of this entry in a list of entries that are used to define a valid line of assembly source code text.

The remainder of the class has two overridden print methods that provide a customized output for this particular class. We're not going to cover the print methods in any greater detail. Now that we've taken a look at the class' code, let's see how the class is used.

## Demonstration: `JsonObjIsValidLineEntry.java`

The `JsonObjIsValidLineEntry` class is a sub-object and the example we've chosen is a utility method from the `AssemblerThumb` class, shown in the subsequent listing.

*Listing 13-7. `JsonObjIsValidLineEntry.java` Demonstration*

File: AssemblerThumb.java

Method: FindValidLineEntryWithMatchingTokenType

Snippet:

```

01 public int[] FindValidLineEntryWithMatchingTokenType(JsonObjIsValidLine validLine,
Token token, int entry, int index) {
02     int count = 0;
03     for (JsonObjIsValidLineEntry validLineEntry :
validLine.is_valid_line) {
04         for (String validLineEntryType : validLineEntry.is_entry_types) {
05             if (token.type_name.equals(validLineEntryType)) {
06                 if (count >= entry) {
07                     return new int[] { count, validLineEntry.index };
08                 } else {
09                     break;
10                 }
11             }
12         }
13         count++;
14     }
15     return null;
16 }
```

A demonstration showing the JsonObjIsValidLineEntry class in use.

The previous example shown is from the `AssemblerThumb` class' `FindValidLineEntryWithMatchingTokenType` method. This method is used to find a valid line entry object for the given `Token`. The method also takes into consideration which valid line entry, `entry` argument, to start searching on. Similarly, the `index` argument, although not currently in use, defines the index to start searching for an entry type match.

The example method is straightforward. We iterate over the valid line object's valid line entry sub-objects and for each one we attempt to find a matching entry type name, line 5. If so the position, `count`, of the valid line and the matching valid line entry are returned as an array of two integers on line 7. If not, the method returns null. That brings us to the conclusion of this demonstration section and the end of the Thumb-1 holder class review. In the next section we'll introduce you to the thumb instruction set's loader classes.

# Thumb-1 Loader Classes

The ARM Thumb-1 instruction set implementation uses a number of loader classes to support bringing data into the GenAsm assembler by parsing and loading a specified list of JSON data files, including the contained objects and sub-objects.

*Listing 13-8. ARM Thumb-1 Specific Loader Classes*

Class	Type	JSON File (If Applicable)
LoaderIsArgTypes.java	List	is_arg_types.json
LoaderIsDirectives.java	List	is_directives.json
LoaderIsEmptyDataLines.java	List	is_empty_data_lines.json
LoaderIsEntryTypes.java	List	is_entry_types.json
LoaderIsOpCodes.java	List	is_op_codes.json
LoaderIsRegisters.java	List	is_registers.json
LoaderIsValidLines.java	List	is_valid_lines.json

A list of the loader classes used to implement the ARM Thumb-1 instruction set.

Note that each JSON file contains different objects that are all loaded in as part of the data loading process.

The complete group of Thumb-1 instruction set specific loader classes, type list, are shown in the previous listing. Note that these classes implement the `Loader` interface directly. In this way they differ slightly from the holder classes we've just finished reviewing. If you recall from our experience with Thumb-1 loader classes, they all extend the `JsonObjBase` class which implements the `JsonObj` interface.

In this case we are missing the centralization that the `JsonObjBase` class provides to the holder classes. As it turns out this level of abstraction is not necessary for the loader classes because they don't have robust class fields or an internal state. That is to say the loader classes populate an instance of a holder class using data from a JSON data file.

Due to the redundant nature of the loader classes, we'll only review two of them in detail. With regard to the rest, I'll leave it to you to take a look at them on your own. Let's list the classes that we're going to review in this section.

- **LoaderIsOpCodes.java**: The loader class responsible for parsing and loading the Thumb-1 instruction set's opcode data file.
- **LoaderIsEntryTypes.java**: The loader class responsible for parsing and loading the Thumb-1 instruction set's list of valid entry types.

The two classes we chose for the review process in this section are the loader classes responsible for loading the opcodes and entry type data files. We'll work with the `LoaderIsOpCodes` class first. Let's take a look at some code.

## Class Review: `LoaderIsOpCodes.java`

The `LoaderIsOpCodes` class is an important loader class that's responsible for loading up the instruction set's opcode definitions. This class is a little larger than previous classes we've reviewed in this chapter, so we'll employ the more robust class review methodology. For this class review we'll look at the following sections:

- Class Fields
- Pertinent Method Outline/Class Headers
- Support Method Details
- Main Method Details
- Demonstration

And with that let's jump into some code!

## Class Fields: `LoaderIsOpCodes.java`

In the first review section we'll take a look at the class' fields. This class has a very concise list of fields so this part of the review will go quickly.

***Listing 13-8. LoaderIsOpCodes.java Class Fields 1***

```
public String obj_name = "LoaderIsOpCodes";
```

The fields of the LoaderIsOpCodes class.

The class has only one field to discuss, the `obj_name` field. This field is added for completeness to match the design pattern of the loader classes and to present some consistent connection between JSON data representations of classes and the associated Java class. In the next section we'll take a look at the class' pertinent methods and class declaration.

## **Pertinent Method Outline/Class Headers: LoaderIsOpCodes.java**

A list of the `LoaderIsOpCodes` class' pertinent methods are as follows.

***Listing 13-9. LoaderIsOpCodes.java Pertinent Method Outline/Class Headers 1***

```
//Support Methods
public JsonObjIsOpCodes ParseJson(String json, String targetClass, String
fileName);

//Main Methods
private void RecursiveSubArgProcessing(List<JsonObjIsOpCodeArg> sub_args,
String fileName);
```

A list of the class' pertinent main and support methods.

The class' header shows import statements and the class' declaration including any super classes it extends or interfaces it implements. Look over the code and be sure that it makes sense to you and that you're familiar with the imports used.

***Listing 13-10. LoaderIsOpCodes.java Pertinent Method Outline/Class Headers 2***

```
package net.middlemind.GenAsm.Loaders.Thumb;
```

```

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import java.lang.reflect.InvocationTargetException;
import java.util.List;
import net.middlemind.GenAsm.Exceptions.Thumb.ExceptionLoader;
import net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsOpCode;
import net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsOpCodeArg;
import net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsOpCodes;
import net.middlemind.GenAsm.Loaders.Loader;

public class LoaderIsOpCodes implements Loader {}

```

The declaration of the `LoaderIsOpCodes` class.

There's not much to cover in this section. Notice that this class implements the `Loader` interface. In the next section we'll cover the `LoaderIsOpCodes` class' support methods.

## Support Method Details: `LoaderIsOpCodes.java`

The opcode loader class has only one support method for us to review. I'll list it here. Take a moment to look it over before we review it.

*Listing 13-11. `LoaderIsOpCodes.java` Support Method Details*

```

01 private void RecursiveSubArgProcessing(List<JsonObjIsOpCodeArg>
sub_args, String fileName) {
02     if (sub_args != null) {
03         for (JsonObjIsOpCodeArg llentry : sub_args) {
04             llentry.name = llentry.getClass().getName();
05             llentry.fileName = fileName;
06             llentry.loader = getClass().getName();
07             llentry.bit_series.name =
llentry.bit_series.getClass().getName();
08             llentry.bit_series.fileName = fileName;
09             llentry.bit_series.loader = getClass().getName();
10
11             if (llentry.num_range != null) {

```

```

12         llentry.num_range.name =
13         llentry.num_range.getClass().getName();
14         llentry.num_range.fileName = fileName;
15         llentry.num_range.loader = getClass().getName();
16     }
17
18     if (llentry.bit_shift != null) {
19         llentry.bit_shift.name =
20         llentry.bit_shift.getClass().getName();
21         llentry.bit_shift.fileName = fileName;
22         llentry.bit_shift.loader = getClass().getName();
23     }
24
25 }
26 }
```

The support methods of the LoaderIsOpCodes class.

The LoaderIsOpCodes class has one support method for us to look at, the RecursiveSubArgProcessing method. This method is used, as you may have guessed, to recursively process sub-objects. Sub-objects, are those that belong to a Token and not the TokenLine. Such a situation occurs during groups, lists, and comments. In our case the opcode definition also has sub-objects. Let's take a look at the method.

On line 2 if the sub-object list is not null, we proceed to iterate over the list, line 3. For each entry we set the name, fileName, and loader class fields, and we set the name, fileName, and loader fields for the entry's bit\_series field, lines 4 – 9. Then on lines 11 – 21 we check to see if the bit\_shift and num\_range fields are defined and if so, we update the same three fields.

This is done to mark each class so that we can trace where it came from when we view the class contents in JSON format. This happens when we use print methods to display the contents of a class instance. The last thing left to do in the method is to check to see if the current sub-object entry has arguments of its own, line 23. This is what makes the method recursive, it

will automatically traverse a nested series of sub-objects. That brings us to the conclusion of this section. In the next section we'll take a look at the class' main methods.

## Main Method Details: LoaderIsOpCodes.java

In this section we'll take a look at the class' main methods. We only have one main method to review, the `JsonObjIsOpCodes` method, listed subsequently.

*Listing 13-12. LoaderIsOpCodes.java Main Method Details*

```
01 public JsonObjIsOpCodes ParseJson(String json, String targetClass,
02                                     String fileName) throws ExceptionLoader {
03     GsonBuilder builder = new GsonBuilder();
04     builder.setPrettyPrinting();
05
06     try {
07         JsonObjIsOpCodes jsonObj =
08             (JsonObjIsOpCodes)Class.forName(targetClass).getConstructor().newInstance();
09
10         jsonObj.name = targetClass;
11         jsonObj.fileName = fileName;
12         jsonObj.loader = getClass().getName();
13
14         jsonObj.bit_series.name =
15             jsonObj.bit_series.getClass().getName();
16         jsonObj.bit_series.fileName = fileName;
17         jsonObj.bit_series.loader = getClass().getName();
18
19         for (JsonObjIsOpCode entry : jsonObj.is_op_codes) {
20             entry.name = entry.getClass().getName();
21             entry.fileName = fileName;
22             entry.loader = getClass().getName();
23             entry.bit_rep.name = entry.bit_rep.getClass().getName();
24             entry.bit_rep.fileName = fileName;
25             entry.bit_rep.loader = getClass().getName();
26             entry.bit_series.name = entry.bit_rep.getClass().getName();
27             entry.bit_series.fileName = fileName;
```

```

26     entry.bit_series.loader = getClass().getName();
27
28     if (entry.args != null) {
29         for (JsonObjIsOpCodeArg lentry : entry.args) {
30             lentry.name = lentry.getClass().getName();
31             lentry.fileName = fileName;
32             lentry.loader = getClass().getName();
33             lentry.bit_series.name =
34                 lentry.bit_series.getClass().getName();
35             lentry.bit_series.fileName = fileName;
36             lentry.bit_series.loader = getClass().getName();
37
38             if (lentry.num_range != null) {
39                 lentry.num_range.name =
40                     lentry.num_range.getClass().getName();
41                 lentry.num_range.fileName = fileName;
42                 lentry.num_range.loader = getClass().getName();
43             }
44
45             if (lentry.bit_shift != null) {
46                 lentry.bit_shift.name =
47                     lentry.bit_shift.getClass().getName();
48                 lentry.bit_shift.fileName = fileName;
49                 lentry.bit_shift.loader = getClass().getName();
50
51             if (lentry.sub_args != null) {
52                 RecursiveSubArgProcessing(lentry.sub_args, fileName);
53             }
54         }
55
56     return jsonObj;
57 } catch (ClassNotFoundException | NoSuchMethodException |
58 InstantiationException | IllegalAccessException |
59 InvocationTargetException e) {
60     throw new ExceptionLoader("Could not find target class, " +
61 targetClass + ", in loader " + getClass().getName());
62 }
63 }
```

The main methods of the LoadIsOpCodes class.

The method is a long one so let's get to it. On lines 2 – 5 we create a new `Gson` object instance, `gson`, to parse the JSON data file. On line 7 we create a new instance of the target class which must extend the `JsonObjIsOpCodes` class. Next, on line 8 the class is populated and on lines 9 – 11 the core field values are set. Remember this class is used to load all of the instruction set opcodes contained in the associated JSON data file.

On lines 13 – 15 we update the core fields of the `JsonObjIsOpCodes` class, the `name`, `fileName`, and `loader` fields. This ensures that any JSON formatted output of this class is recognizable and traceable back to the data file and loading class. The loop on lines 17 – 54 is responsible for iterating over each opcode sub-object and setting its core fields. This includes checking for bit series, number range, bit shift, and opcode argument objects while also updating their fields.

This ensures full traceability when looking over JSON output files from the assembly process. On lines 18 – 26 the core fields are set on the opcode object and any associated sub-objects. On lines 28 and 29 if the opcode argument has sub-objects, we iterate over them. On lines 30 – 35 the core fields are set for the opcode argument object and its bit series sub-object.

On lines 37 – 47 we set the core fields for the number range and bit shift sub-objects if they're defined. Lastly on lines 49 – 51, and this is where the support method we reviewed comes into play, if the opcode argument sub-object has arguments of its own, we fire off a recursive call to update core fields on those sub-objects as well. That brings us to the end of this review section. In the next section we'll take a look at the class in use.

## Demonstration: LoaderIsOpCodes.java

The demonstration of the `LoaderIsOpCodes` class will be brief. We'll take a look at the class as it's used by the `AssemblerThumb` class as part of the assembly process.

*Listing 13-13. LoaderIsOpCodes.java Demonstration*

File: `AssemblerThumb.java`

Method: LoadAndParseJsonObjData

Snippet:

```
01 Class cTmp = null;
02 Loader ldr = null;
03 String json = null;
04 String jsonName = null;
05 JsonObject jsonObj = null;
06
07 for(JsonObjIsFile entry : isaDataSet.is_files) {
08     if(eventHandler != null) {
09         eventHandler.LoadAndParseJsonObjDataLoopPre(step, this, entry);
10     }
11
12     cTmp = Class.forName(entry.loader_class);
13     ldr = (Loader)cTmp.getDeclaredConstructor().newInstance();
14     json = null;
15     jsonName = null;
16     jsonObj = null;
17
18     isaLoader.put(entry.loader_class, ldr);
19     Logger.wrl("AssemblerThumb: RunAssembler: Loader created '" +
entry.loader_class + "'");
20
21     json = FileLoader.LoadStr(entry.path);
22     jsonSource.put(entry.path, json);
23     Logger.wrl("AssemblerThumb: RunAssembler: Json loaded '" +
entry.path + "'");
24
25     jsonObj = ldr.ParseJson(json, entry.target_class, entry.path);
26     Logger.wrl("AssemblerThumb: RunAssembler: " + jsonObj.GetName());
27     jsonName = jsonObj.GetName();
28     isaData.put(jsonName, jsonObj);
29     Logger.wrl("AssemblerThumb: RunAssembler: Json parsed as '" +
entry.target_class + "'");
30     Logger.wrl("AssemblerThumb: RunAssembler: Loading isaData with
entry '" + jsonName + "'");
31
32
33 if(jsonObj.GetLoader().equals("net.middlemind.GenAsm.Loaders.Thumb.LoaderI
sEntryTypes")) {
34     jsonObjIsEntryTypes = (JsonObjectIsEntryTypes)jsonObj;
35     Logger.wrl("AssemblerThumb: RunAssembler: Found
JsonObjectIsEntryTypes object, storing it...");
```

```

35
36     } else
37     if(jsonObj.GetLoader().equals("net.middlemind.GenAsm.Loaders.Thumb.LoaderI
sValidLines")) {
38         jsonObjIsValidLines = (JsonObjIsValidLines)jsonObj;
39         Logger.wrl("AssemblerThumb: RunAssembler: Found
JsonObjIsValidLines object, storing it...");
```

40 } else
41 if(jsonObj.GetLoader().equals("net.middlemind.GenAsm.Loaders.Thumb.LoaderI
sEmptyDataLines")) {
42 jsonObjIsEmptyDataLines = (JsonObjIsEmptyDataLines)jsonObj;
43 Logger.wrl("AssemblerThumb: RunAssembler: Found
JsonObjIsEmptyDataLines object, storing it...");

44 } else
45 if(jsonObj.GetLoader().equals("net.middlemind.GenAsm.Loaders.Thumb.LoaderI
sOpCodes")) {
46 jsonObjIsOpCodes = (JsonObjIsOpCodes)jsonObj;
47 lineBitSeries = jsonObjIsOpCodes.bit\_series;

48 lineNumRange = new JsonObjNumRange();
49 lineNumRange.alignment = "WORD";
50 lineNumRange.bcd\_encoding = false;
51 lineNumRange.bit\_len = lineBitSeries.bit\_len;
52 lineNumRange.obj\_name = "JsonObjNumRange";
53 lineNumRange.min\_value = 0;
54 lineNumRange.max\_value = 65536;
55 lineNumRange.ones\_complement = false;
56 lineNumRange.twos\_complement = false;

57
58 pcPrefetchBytes = jsonObjIsOpCodes.pc\_prefetch\_bytes;
59 pcPrefetchHalfwords = jsonObjIsOpCodes.pc\_prefetch\_halfwords;
60 pcPrefetchWords = jsonObjIsOpCodes.pc\_prefetch\_words;

61
62
63 if(jsonObjIsOpCodes.endian.equals(AssemblerThumb.ENDIAN\_NAME\_BIG)) {
64 isEndianBig = true;
65 isEndianLittle = false;
66 } else {
67 isEndianBig = false;
68 isEndianLittle = true;
69 }

```

69         Logger.wrl("AssemblerThumb: RunAssembler: Found
JsonObjIsOpCodes object, storing it...");  

70  

71     } else  

72     if(jsonObj.GetLoader().equals("net.middlemind.GenAsm.Loaders.Thumb.LoaderI  

sDirectives")) {  

73         jsonObjIsDirectives = (JsonObjIsDirectives)jsonObj;  

74         Logger.wrl("AssemblerThumb: RunAssembler: Found  

JsonObjIsDirectives object, storing it...");  

75     } else  

76     if(jsonObj.GetLoader().equals("net.middlemind.GenAsm.Loaders.Thumb.LoaderI  

sRegisters")) {  

77         jsonObjIsRegisters = (JsonObjIsRegisters)jsonObj;  

78         Logger.wrl("AssemblerThumb: RunAssembler: Found  

JsonObjIsRegisters object, storing it...");  

79     }  

80     if(eventHandler != null) {  

81         eventHandler.LoadAndParseJsonObjDataLoopPost(step, this,  

entry);  

82     }  

83 }

```

An example of the LoaderIsOpCodes class in use. This listing shows the AssemblerThumb class' data loading loop.

The demonstration of the LoaderIsOpCodes class, shown previously, is from the AssemblerThumb class' LoadAndParseJsonObjData method. This method is responsible for loading all the necessary data to begin assembling the provided source code file. On lines 1 – 5 the loop variables are initialized and on line 7 we iterate over the files associated with this instruction set.

On lines 8 – 10, if defined, we call the specified callback method. The callback method is used to provide some level of customization as you can implement it and use it as an opportunity to customize the process. On lines 12 – 16 we initialize the loop variables for this iteration of the loop. Notice on

lines 12 – 13 we initialize the loader from a data driven value. The only requirement is that the target loader class implements the `Loader` interface.

Lines 18 and 19 register the loader class in the `isaLoader` field and logs the occurrence. Next up, on lines 21 – 23, we get a copy of the JSON data file's source and store it in the `jsonSource` field with logging. On lines 25 and 26 the dynamically created loader parses the specified JSON data file and creates the new holder class instance with logging.

Next, on lines 27 – 30 the name of the JSON data object is stored in a local variable, `jsonName`, the holder object is then stored in the `isaData` field, and the occurrence is thoroughly logged. On lines 32 to 78 we check to see which data file has been loaded and store it in a direct field for quick access. There are some custom steps performed for each file. We won't review this in detail here as it's slightly outside the scope of this section. Take a moment to review it on your own.

Notice the closing callback method call on lines 80 – 82. These callback methods give you further chances to customize the assembly process although they aren't used in this assembler implementation. That brings us to the end of this class review. In the next section we'll take a close look at another loader class to round out our review.

## Class Review: LoaderIsEntryTypes.java

The next loader class we'll review in this section is the `LoaderIsEntryTypes` class. This class is important because it's responsible for loading up the entry types supported by a given instruction set. The class is concise, so we'll review it in one listing. Let's take a look.

*Listing 13-14. LoaderIsEntryTypes.java Full Class Review*

```
01 package net.middlemind.GenAsm.Loaders.Thumb;
02
03 import com.google.gson.Gson;
04 import com.google.gson.GsonBuilder;
05 import java.lang.reflect.InvocationTargetException;
06 import net.middlemind.GenAsm.Exceptions.Thumb.ExceptionLoader;
07 import net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsEntryType;
08 import net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsEntryTypes;
```

```

09 import net.middlemind.GenAsm.Loaders.Loader;
10
11 public class LoaderIsEntryTypes implements Loader {
12     public String obj_name = "LoaderIsEntryTypes";
13
14     @Override
15     public JsonObjIsEntryTypes ParseJson(String json, String
targetClass, String fileName) throws ExceptionLoader {
16         GsonBuilder builder = new GsonBuilder();
17         builder.setPrettyPrinting();
18
19         Gson gson = builder.create();
20         try {
21             JsonObjIsEntryTypes jsonObj = (JsonObjIsEntryTypes)
Class.forName(targetClass).getConstructor().newInstance();
22             jsonObj = gson.fromJson(json, jsonObj.getClass());
23             jsonObj.name = targetClass;
24             jsonObj.fileName = fileName;
25             jsonObj.loader = getClass().getName();
26
27             for (JsonObjIsEntryType entry : jsonObj.is_entry_types) {
28                 entry.name = entry.getClass().getName();
29                 entry.fileName = fileName;
30                 entry.loader = getClass().getName();
31                 entry.txt_match.name =
entry.txt_match.getClass().getName();
32                 entry.txt_match.fileName = fileName;
33                 entry.txt_match.loader = getClass().getName();
34             }
35
36             return jsonObj;
37         } catch (ClassNotFoundException | NoSuchMethodException |
InstantiationException | IllegalAccessException | InvocationTargetException e) {
38             throw new ExceptionLoader("Could not find target class, " +
targetClass + ", in loader " + getClass().getName());
39         }
40     }
41 }
```

A full listing of the LoaderIsEntryTypes class.

Much of this class should look familiar to you. Take a moment to look at the import statements used by the class, lines 3 – 9. Notice that the class implements the `Loader` interface, line 11. On line 12 the requisite `obj_name` field is defined and defaulted to the name of the class. On lines 15 – 40 the `ParseJson` method is defined.

This method should be familiar from previous chapters, so we won't go into too much detail here. The basic responsibility of this method is to load the JSON data file into a holder class and to set the core fields `name`, `fileName`, `loader`, as we've seen previously. In the next section we'll take a look at the class in action.

## Demonstration: LoaderIsEntryTypes.java

The demonstration of the `LoaderIsEntryTypes` class is actually part of the demonstration from the preceding class review.

*Listing 13-15. LoaderIsEntryTypes.java Demonstration*

File: `AssemblerThumb.java`  
Method: `LoadAndParseJsonObjData`  
Snippet:

```
1
if(jsonObj.GetLoader().equals("net.middlemind.GenAsm.Loaders.Thumb.LoaderI
sEntryTypes")) {
2     jsonObjIsEntryTypes = (JsonObjIsEntryTypes)jsonObj;
3     Logger.wrl("AssemblerThumb: RunAssembler: Found jsonObjIsEntryTypes
object, storing it...");
```

4 }

An example of the `LoaderIsEntryTypes` class in use. This listing is from the `AssemblerThumb` class' data loading loop.

Since the file loading is generalized and data driven it is not clear in the code when this loader is executed. The demonstration above shows the detection of this loader class in the `AssemblerThumb` class' data file loading loop. The object is stored in the `jsonObjIsEntryTypes` class field for quick

access and the event is logged. That brings us to the conclusion of this class review and of the loader class section. In the next section we'll take a look at the Thumb-1 exception classes.

## Thumb-1 Exception Classes

The Thumb-1 exception classes are used to help customize and define the errors that occur during the assembly process. There are a number of custom exception classes. They all follow the same pattern. Take a moment to review the ones that we don't cover here. The two classes that we'll take a look at are as follows.

- **ExceptionJsonObjLink:** A custom exception class used during object linking, the connection between JSON data driven text and actual Java objects.
- **ExceptionLoader:** A custom exception class used during JSON data file loading.

These classes are very short, so we'll review the class in one listing. You'll find the demonstration section equally concise. Let's take a look.

### Class Review: `ExceptionJsonObjLink.java`

The `ExceptionJsonObjLink` class is used when reporting exceptions during the object linking phase of JSON data file loading.

*Listing 13-16. `ExceptionJsonObjLink.java` Full Class Review*

```
1 package net.middlemind.GenAsm.Exceptions.Thumb;
2
3 import net.middlemind.GenAsm.Exceptions.ExceptionBase;
4
5 public class ExceptionJsonObjLink extends ExceptionBase {
6     public ExceptionJsonObjLink(String errorMEssage) {
7         super(errorMEssage);
8     }
9 }
```

A full listing of the ExceptionJsonObjLink class.

Notice in the class listing on line 5 that the class extends the ExceptionBase class. All exceptions for the Thumb-1 instruction set are setup this way. There's not much to the class except the constructor that takes a custom error message as an argument. Let's see the class in action.

## Demonstration: ExceptionJsonObjLink.java

To demonstrate the ExceptionJsonObjLink class we'll look at the JsonObjIsArgTypes class' Link method. This method is used to create a reference to an object by searching a list of loaded objects, in this case JsonObjIsEntryType instances, to find an object with a matching name.

*Listing 13-17. ExceptionJsonObjLink.java Demonstration*

File: JsonObjIsArgTypes.java  
Method: Link  
Snippet:

```
01 for(String s : entry.is_entry_types) {  
02     boolean found = false;  
03     for (JsonObjIsEntryType lentry : ((JsonObjIsEntryTypes)  
linkData).is_entry_types) {  
04         if (!Utils.IsEmpty(lentry.type_name) &&  
lentry.type_name.equals(s)) {  
05             entry.linked_is_entry_types.add(lentry);  
06             found = true;  
07             break;  
08         }  
09     }  
10     if (!found) {  
11         throw new ExceptionJsonObjLink("JsonObjIsArgTypes: Link: Error:  
Could not find JsonObjIsEntryType, group or single, object with name " +  
s);  
13     }  
14 }
```

A demonstration of the `ExceptionJsonObjLink` class in use with the `JsonObjIsArgTypes` class' `Link` method.

On line 1 of the demonstration listing we iterate over the JSON based linking data, the string values in the `is_entry_types` array, and search for a matching Java object instance with the same name, line 3. The check is performed on line 4 and if a match is found the `JsonObjIsEntryType` object instance is registered in the `linked_is_entry_types` field. This establishes an actual connection between the two objects and the order of the argument's entry types is preserved.

However, if no match is found, lines 11 – 13, then we need to push the panic button because we are referencing an entry type that doesn't exist and this can cause big problems down the road. To handle this correctly we need to throw an exception. On line 12 we throw a new `ExceptionJsonObjLink` exception with some details about the error we encountered. That's all I wanted to cover in this demonstration section. In the next section we'll take a look at one more Thumb-1 exception class, the `ExceptionLoader` class.

## Class Review: `ExceptionLoader.java`

The `ExceptionLoader` class is very similar to the custom exception class we've just reviewed. The class is listed as follows.

*Listing 13-18. `ExceptionLoader.java` Full Class Review*

```
1 package net.middlemind.GenAsm.Exceptions.Thumb;
2
3 import net.middlemind.GenAsm.Exceptions.ExceptionBase;
4
5 public class ExceptionLoader extends ExceptionBase {
6     public ExceptionLoader(String errorMessage) {
7         super(errorMessage);
8     }
9 }
```

A full listing of the `ExceptionLoader` class.

Note that the same super class is extended, `ExceptionBase`, and the constructor takes the same string based, error message, argument. There's not much more to discuss here. Let's take a look at this class in use.

## Demonstration: `ExceptionLoader.java`

The demonstration of the `ExceptionLoader` class is from the `LoaderIsSets` class' `ParseJson` method. The example is listed as follows.

*Listing 13-19. `ExceptionLoader.java` Demonstration*

File: `LoaderIsSets.java`

Method: `ParseJson`

Snippet:

```
1 } catch (ClassNotFoundException | NoSuchMethodException |  
InstantiationException | IllegalAccessException |  
InvocationTargetException e) {  
2     throw new ExceptionLoader("Could not find target class, " +  
targetClass + ", in loader " + getClass().getName());  
3 }
```

A demonstration of the `ExceptionLoader` class in use.

Take a moment to note the different exceptions that this catch clause is configured to respond to. In each case the exception indicates that the JSON parse operation has failed, and the target class has not been populated. To indicate this error a new `ExceptionLoader` exception is thrown. This approach centralizes and simplifies the exception for other assembler code that catches it. That's all we have to say about the demonstration of this exception class. And with that we have reached the end of this class review section and the chapter.

## Chapter Conclusion

In this chapter we took a look at the foundation of the Thumb-1 instruction set implementation in the GenAsm assembler project. This approach led us to

look at Thumb-1 specific implementations of loader, holder, and exception classes. Let's list the classes we've reviewed in this chapter here.

- **JsonObjIsArgTypes.java**: Demonstrates a list holder class.
- **JsonObjIsRegister.java**: Demonstrates an object holder class.
- **JsonObjIsValidLineEntry.java**: Demonstrates a sub-object holder class.
- **LoaderIsOpCodes.java**: The loader class responsible for parsing and loading the Thumb-1 instruction set's opcode data file.
- **LoaderIsEntryTypes.java**: The loader class responsible for parsing and loading the Thumb-1 instruction set's list of valid entry types.
- **ExceptionJsonObjLink**: A custom exception class used during object linking, the connection between JSON data driven text and actual objects.
- **ExceptionLoader**: A custom exception class used during JSON data file loading.

We've looked over a lot of material in preparation for the review of the GenAsm assembler's code. This chapter was the last preparation chapter. Following this point, we'll be looking at the gory details of an ARM Thumb-1 assembler implementation. If you've made it this far then you definitely deserve a pat on the back. Take a deep breath and get ready for the next chapter in this project review.

# Chapter 14: The Preprocessor

Now that we've finished a lengthy review of the underlying material, including but not limited to the following topics, listed subsequently, we can begin looking at the ARM Thumb-1 specific code that powers this assembler. Before we move on, however, let's summarize the main topics we've covered thus far.

## Topics Covered

- **The Thumb-1 Instruction Set:** Detailed review of the opcodes involved with the target instruction set for our implementation.
- **The Preprocessor and Standard Directives:** A complete review of the directives supported by the GenAsm assembler.
- **Base Class Review:** Detailed review of the supporting classes that provide the foundation for much of the project's functionality.
- **JSON Data File Review:** A review of the JSON data files used to contain all of the ARM Thumb-1 instruction set data the assembler needs to do its job.

- **Static Main Entry Point – Program Execution:** We reviewed the program's command line arguments and how to execute the program.
- **The Thumb-1 Implementation of Base Classes:** We reviewed the Thumb-1 implementation of the project's base classes in preparation for a review of the Thumb-1 implementation's core code.

As you can see, we've covered a lot of material up to this point in the text. Next let's take a look at the remaining topics for us to discuss.

## Topics Remaining

- **The Preprocessor:** The first step in the assembly process that occurs before all JSON data loading and any source file parsing. Supports a few directives that replace text in the assembly source code file.
- **The Lexer:** The second step in the assembly process that occurs after the preprocessor runs. This step is responsible for pulling out pieces of text from the assembly source code while ignoring white space.
- **The Tokenizer:** The third step in the assembly process tokenizes the string artifacts, identifying them as a specific entry type. At this point we're ready to begin assembling the source code.
- **The Assembler:** The main step in the assembly process. The assembler converts the assembly source code into an internal data structure that is then compared to the instruction set data files. If the opcodes and arguments match up the assembler can convert that line of source code to a binary representation.
- **The Linker:** The last step in the process. The linker is responsible for connecting different areas of assembly source code together along with the associated binary representation to create the listing files and the final binary output files.

The remaining topics up for us to review are listed previously. We're a little more than halfway through the material at this point in the text. In the

next few chapters, we'll be looking under the hood at the Thumb-1 implementation's core code. And with that let's begin our review of the Thumb-1 preprocessor class.

## Class Review: PreProcessorThumb.java

To cover this class, we'll use the more detailed class review protocol and touch upon the following sections.

- Static/Constants/Read-Only Class Members
- Class Fields
- Pertinent Method Outline/Class Headers
- Support Method Details
- Main Method Details
- Demonstration

There are no pertinent enumerations to speak of, so we'll omit that review section. Let's start things off with a review of the static class members.

### Static/Constants/Read-Only Class Members: PreProcessorThumb.java

The `PreProcessorThumb` class contains a number of static class fields that are used to represent some key aspects of the preprocessor.

*Listing 14-1. PreProcessorThumb.java Static/Constants/Read-Only Class Members*

```
public static String[] PP_DIRECTIVES = { "$INCBIN", "$INCASM", "$NOP",
"$STRING", "$FLPINCBIN", "$FLIPSTRING" };

public static int PPD_INCBIN_IDX = 0;
public static int PPD_INCASM_IDX = 1;
public static int PPD_NOP_IDX = 2;
public static int PPD_STRING_IDX = 3;
public static int PPD_FLPINCBIN_IDX = 4;
public static int PPD_FLIPSTRING_IDX = 5;
public static byte PPD_EOL_BYTE = (byte)255;
```

```
public static String OUTPUT_FILE_NAME =
"output_pre_processed_assembly.txt";

public static String[] EXT_ASM_FILES = { ".asm", ".txt", ".s" };
public static String[] EXT_BIN_FILES = { ".bin", ".raw", ".dat" };

public static int MAX_EXT_BIN_FILE_LEN = 255;
public static int MAX_STRING_LEN = 32;
```

The static class fields of the PreProcessorThumb class.

The PreProcessorThumb class has a few static class fields for us to look at. In the previous listing, the first static class field shown is the PP\_DIRECTIVES field. This is an array that holds the case-sensitive name of each directive supported by the preprocessor. The next six fields, PPD\_INCBIN\_IDX to PPD\_FLIPSTRING\_IDX, are used as quick access index values. These fields can be used to reference a preprocessor directive in the PP\_DIRECTIVES array in a standardized and fast way.

The subsequent field, PPD\_EOL\_BYTE, is a static field that holds a byte value used to indicate the end of a string value. The next field listed, OUTPUT\_FILE\_NAME, holds the name of output file for this step in the assembly process. The next two fields EXT\_ASM\_FILES and EXT\_BIN\_FILES are static arrays that list the file extensions of the supported file types that can be used with the include assembly and include binary preprocessor directives.

The last two fields in the set are used to limit some of the preprocessor's directives. For example, the MAX\_EXT\_BIN\_FILE\_LEN field has a value of 255, meaning that the maximum size, currently, of an external binary file is 255 bytes. That's actually not too bad but you may want to increase this value. The last static field is the MAX\_STRING\_LEN field and it's used to limit the length of strings that are handled by the preprocessor to 32 characters. Again, this is fairly concise so you can increase it a little as needed.

## Class Fields: PreProcessorThumb.java

In this section we'll take a look at the `PreProcessorThumb` class' fields. There are only three class fields to look at, listed subsequently.

*Listing 14-2. PreProcessorThumb.java Class Fields 1*

```
public String asmSourceFile = "";
public String rootOutputDir = "";
public Object other = null;
```

The static class fields of the `PreProcessorThumb` class.

The first class field, `asmSourceFile`, holds the full path to the original assembly source code file specified. Recall from our review of the project's static main entry point that the assembly source code file is specified as part of the program's required argument set. The next class field, `rootOutputDir`, holds the path to the output directory where the output file should be written. The last field in the list is a generic Java object, `other`, this object can be provided to the preprocessor as a customization but currently it's not used by this implementation. In the next section we'll take a look at the class' pertinent methods and declaration.

## Pertinent Method Outline/Class Headers: PreProcessorThumb.java

The `PreProcessorThumb` class is concise. There are only two pertinent methods for us to review.

*Listing 14-3. PreProcessorThumb.java Pertinent Method Outline/Class Headers 1*

```
//Main Methods
public List<String> RunPreProcessor(String assemblySourceFile, String
outputDir, Object otherObj);

//Support Methods
public void AddPrefix(List<String> lines, String appendix);
```

A listing of the PreProcessorThumb class' pertinent methods.

The first method listed is a required implementation defined by the PreProcessor interface. The second method listed is a support method that helps add a prefix, usually indentation spacing, to a line of assembly source code. Let's take a look at the class' declaration next.

*Listing 14-4. PreProcessorThumb.java Pertinent Method Outline/Class Headers 2*

```
package net.middlemind.GenAsm.PreProcessors.Thumb;

import java.io.File;
import java.nio.ByteBuffer;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.Hashtable;
import java.util.List;
import java.util.Map;
import
net.middlemind.GenAsm.Exceptions.Thumb.ExceptionUnsupportedAssemblyType;
e;

import
net.middlemind.GenAsm.Exceptions.Thumb.ExceptionUnsupportedBinaryLength;
h;

import
net.middlemind.GenAsm.Exceptions.Thumb.ExceptionUnsupportedStringLength;

import net.middlemind.GenAsm.FileIO.FileLoader;
import net.middlemind.GenAsm.FileIO.FileUnloader;
import net.middlemind.GenAsm.Logger;
import net.middlemind.GenAsm.PreProcessors.PreProcessor;
import net.middlemind.GenAsm.Utils;

public class PreProcessorThumb implements PreProcessor { }
```

The PreProcessorThumb class' declaration.

The `PreProcessorThumb` declaration is in line with what we've seen thus far. Take a moment to review the import statements and notice that the class implements the `PreProcessor` interface. In the next section we'll take a look at the class'.

## Support Method Details: `PreProcessorThumb.java`

The `PreProcessorThumb` class has only one support method for us to review. The `AddPrefix` method is used to store a string associated with a line of assembly source code so that the string value can be prepended to the source code line when the updated assembly source code is written out.

*Listing 14-5. `PreProcessorThumb.java` Support Method Details 1*

```
1 public void AddPrefix(List<String> lines, String prefix) {  
2     for (int i = 0; i < lines.size(); i++) {  
3         lines.set(i, prefix + lines.get(i));  
4     }  
5 }
```

The `PreProcessorThumb` class' support methods.

The `AddPrefix` method takes two arguments, the first is a list of assembly lines and the second argument is the string that will be appended to each line of the `List` provided. This support method is used to add a prefix, in this case the prefix is most likely the proper indent for the line of assembly code. This feature is used to ensure newly added lines of code, the result of preprocessor include directives, have the same indentation as the original line of source code.

## Main Method Details: `PreProcessorThumb.java`

The `PreProcessorThumb` class has one main method for us to review. This method, `RunPreProcessor` listed subsequently, is rather lengthy so we'll break up the method listing into a few parts that loosely correspond to a particular preprocessor directive.

#### *Listing 14-6. PreProcessorThumb.java Main Method Details 1*

The PreProcessorThumb class' main method listing part 1, showing method initialization and the \$NOP preprocessor directive.

On lines 3 – 5 the method arguments are stored in class fields. A large block of method variables are declared on lines 7 – 25. Let's go over them in detail starting with the variable `ret` on line 7. The `ret` variable is short for return and is intended to hold the final version of the assembly source code that is returned at the end of the method. Notice that the `FileLoader` class is used to read the contents of the specified file into a `List` of strings. In this case each string entry is a line of assembly source code.

The local variable, `idxs`, is an integer array that is used in conjunction with the `Utils` class' `StringContainsArrayEntry` method. Recall that this method returns the index in the provided array of the string's match as well as the character index of the match. We'll use this method to find preprocessor directives in a line of assembly source code while tracking which directive was found and at what position in the line it was located.

The next five local variables, lines 9 – 13, are used as temporary variables during the processing of directives. The variable `directive` is used to track which preprocessor directive we're currently working on. The next variable, `sIdx`, is used to reference return values from the `idxs` array. The next two variables, strings `tmp` and `fileName` are used in formatting paths when reading files specified by binary and assembly include preprocessor directives.

The last entry in this group is the `count` variable. This variable is used to track the line number, starting at zero of each line in the assembly source code file. Next up, the `incAsm` variable is a `List` of strings that's used to hold the contents of an included assembly file directive. Can you see why this variable and the `ret` variable are similar? It's because they are both designed to hold assembly source code separated out by line.

The subsequent three variables, `asmFileAdj`, `asmFileAdjTypes`, and `asmFileAdjNames` are all used to track information about the adjustments necessary to add the specified include directive, in-line, into the

current assembly source code. We'll look into these variables in more detail when we cover this method's core functionality. Following these fields are the `File`, `f`, and the string `lcFileName`, lines 18 and 19 respectively. These variables are used for preprocessor directives that require reading a file like the include assembly or the include binary directives.

The next three variables are specific to the include binary directive, lines 20 – 22. The first entry is the `incBin` variable. This variable is an array of bytes and is used to store the binary data in the specified file. The next two integer variables, `numBytes` and `numHalfWords`, are used to track how much binary data is being processed by the include binary directive.

The `paddingOn` Boolean variable is used to toggle binary padding. When this value is set to true, during include data and string directives, the data written is forced to half-word boundaries by padding any missing values with zeros. The last variable listed, `asmFileReplace`, is specific to the `$NOP` preprocessor directive. It's a `Map` variable used to keep track of which lines require the necessary replacement.

That wraps up our review of the local method variables. Not too bad. We are performing some fairly complex manipulation at this stage of the assembly process but nothing we can't handle. On line 27 we iterate over the lines of the provided assembly source code file. If the line is defined, we then process it to find what preprocessor directive, if any, is specified, line 29.

Note that there is an inherent limitation in the way this preprocessor was implemented. Can you see what it is? Our preprocessor can only handle one directive per line. It doesn't have the ability to process multiple directives on one line. This is fine for our purposes, I just wanted to point it out. If a directive has been found, line 30, we convert the integer representation of the preprocessor directive to a string on line 31.

This will come in handy when we write the preprocessor directive headers and include the original line of code commented out. Headers in this context means a brief text description of what directive was processed. On lines 33 – 35 we detect the indentation on the preprocessor directive line. Notice that there is another requirement here? Can you see it? The preprocessor directives should be commented out. That is, they should come after an assembly comment character, `";"`.

This piece of code is used to capture the indent spacing so that it can be restored when the preprocessor directive is replaced. In this way the original assembly source code has a better chance of maintaining its visual structure with regard to indentation. The first preprocessor directive we handle is the `$NOP` directive on line 37. Recall from our review of the assembler's supported preprocessor directives that this directive is used to inject an innocuous line of assembly code into the original source code.

Handling this directive is easy, on line 38 we store the line of the directive and the innocuous line of assembly to inject. We'll see this variable again, `asmFileReplace`, when we output the modified assembly source code. The next directive handled by the `RunPreProcessor` method is the include assembly directive which starts on line 40. I've listed the section of code responsible for this directive subsequently. Look over the code and we'll review it in detail.

*Listing 14-7. PreProcessorThumb.java Main Method Details 2*

```
040 } else if (idxs[0] == PPD_INCASM_IDX) {  
041     sIdx = idxs[1];  
042     tmp = s.substring(s.indexOf("|") + 1, s.lastIndexOf("|")).trim();  
043  
044     fileName = tmp;  
045     f = new File(fileName);  
046     lcFileName = f.getName().toLowerCase();  
047  
048     if (Utils.StringContainsArrayEntry(EXT_ASM_FILES, lcFileName) !=  
null) {  
049         incAsm = FileLoader.Load(fileName);  
050         incAsm.add(0, ";Found file with line count: " + incAsm.size() +  
", byte count: " + f.length());  
051         incAsm.add(1, "");  
052         incAsm.add(incAsm.size(), "");  
053  
054         asmFileAdjTypes.put(s, idxs[0]);  
055         asmFileAdj.put(s, incAsm);  
056         AddPrefix(incAsm, whiteSpace);  
057         asmFileAdjNames.put(s, fileName);  
058  
059     } else {
```

```
060         throw new ExceptionUnsupportedAssemblyFileType("Cannot load  
assembly files without file extension .asm or .txt for file name, " +  
fileName);  
061     }  
062  
063 } ...
```

The PreProcessorThumb class' main method listing part 2, showing the include assembly preprocessor directive.

On line 40 we check to see if the directive found is the include assembly directive by checking that the index values match. Next, on line 41 we store a local copy of the text position of the directive match and on line 42 we parse the directive to extract the specified assembly file to include. As you can see from the code the file is specified in between matching pipe, "|", characters.

The `fileName` variable is set, using the extracted file name, on line 44 and on line 45 we instantiate a new `File` object instance using the extracted file name. We want to check the file's extension to make sure it's supported by the GenAsm assembler. To do so we'll need a lower-case copy of just the file name without the associated full path. This is handled on line 46 and on line 48 we check to see if the specified file has a supported file extension otherwise, we throw an exception, lines 59 – 61.

The code on lines 49 to 57 is responsible for reading the specified assembly file to include, and for prepping a few method variables that we'll use a little later on in the method to write the modified source code. On line 49 the specified assembly source code file to include is loaded into the `incAsm` variable. We want to prefix the included assembly source code with a little header information to describe what's been done and help troubleshoot any issues.

To do this we insert a new value into the `incAsm` list, line 50, and include both the line count and the binary length of the file. We use this same approach to add a line break before and after the included assembly source code. The last bit of code in this section, lines 54 – 57, is used to register some key information about the current directive. On line 54 the line of

assembly source code is used as the key, to a `Hashtable`, while the type of the directive is used as the associated value.

Similarly, on line 55 we use the original line of assembly code as a key and the loaded assembly source code from the specified file as the associated value. Notice that on line 56 there is a call to the `AddPrefix` method we reviewed earlier. This method call ensures that the included assembly source code has the same indentation as the original line of source code which helps to keep things looking nice and orderly.

Lastly, on line 57, we store the original file name, also as the value in an associated key-value pair. Can you see a pattern here? Can you determine why we're using the original line of assembly source code as a key instead of, say, the line number? The reason why we chose to use the actual source code as the key, as opposed to the line number, is because the source code on a given line isn't going to change as directives are processed.

The line number of said assembly source code, however, will change with directives that insert new lines of source code. The best way for us to track the changes we need to make is to associate them with the original line of source code that contains the operative preprocessor directive. In the subsequent listing we take a look at the include binary preprocessor directive. Recall from our review of these directives that the include binary directive is used to insert data directives into the source code for each value in the binary data.

*Listing 14-8. PreProcessorThumb.java Main Method Details 3*

```
063 } else if (idxs[0] == PPD_INCBIN_IDX || idxs[0] == PPD_FLPINCBIN_IDX)
{
064     sIdx = idxs[1];
065     tmp = s.substring(s.indexOf("|") + 1, s.lastIndexOf("|")).trim();
066
067     fileName = tmp;
068     f = new File(fileName);
069     lcFileName = f.getName().toLowerCase();
070
071     if (Utils.StringContainsArrayEntry(EXT_BIN_FILES, lcFileName) != null) {
072         incBin = FileLoader.LoadBin(fileName);
```

```

073     numBytes = incBin.length;
074     numHalfWords = numBytes / 2;
075     paddingOn = false;
076
077     if (numBytes > MAX_EXT_BIN_FILE_LEN) {
078         throw new ExceptionUnsupportedBinaryFileLength("Cannot load
an external assembly file with length greater than " +
MAX_EXT_BIN_FILE_LEN + " bytes");
079     }
080
081     if (numBytes % 2 == 1) {
082         paddingOn = true;
083     }
084
085     short[] shorts = new short[numHalfWords];
086     ByteBuffer.wrap(incBin).asShortBuffer().get(shorts);
087     List<String> nret = new ArrayList<>();
088     String hex = null;
089     byte[] tIncBin = null;
090     int tCount = 0;
091     String binLine = null;
092     int i = 0;
093     String idxStr = null;
094     nret.add(";Found file with byte count: " + numBytes + ", word
count: " + numHalfWords + ", and padding on: " + paddingOn);
095     nret.add("");
096
097     for (i = 0; i < shorts.length; i++) {
098         hex = Integer.toHexString(shorts[i] & 0xffff);
099         hex = Utils.FormatHexString(hex.toUpperCase(), 4);
100         if (idxs[0] == PPD_FLPINCBIN_IDX) {
101             hex = UtilsEndianFlipHex(hex);
102         }
103         hex = Utils.FormatHexString(hex.toUpperCase(), 4);
104         binLine = Integer.toHexString((tCount * 2) & 0xffff);
105         binLine = Utils.FormatHexString(binLine.toUpperCase(), 4);
106         idxStr = Utils.FormatBinString((i + ""), 4, true);
107
108         nret.add("@DCHW #" + hex + "\t\t;index: " + idxStr + "\\
taddress: " + binLine);
109         tCount++;
110     }
111

```

```

112     i++;
113
114     if (paddingOn) {
115         shorts = new short[1];
116         tIncBin = new byte[]{incBin[incBin.length - 1], 0};
117         ByteBuffer.wrap(tIncBin).asShortBuffer().get(shorts);
118
119         hex = Integer.toHexString(shorts[0] & 0xffff);
120         hex = Utils.FormatHexString(hex.toUpperCase(), 4);
121         if (idxs[0] == PPD_FLPINCBIN_IDX) {
122             Logger.wrl("Flipping bin entry: " + hex);
123             hex = UtilsEndianFlipHex(hex);
124             Logger.wrl("Flipping bin entry: " + hex);
125         }
126         hex = Utils.FormatHexString(hex.toUpperCase(), 4);
127         binLine = Integer.toHexString((tCount * 2) & 0xffff);
128         binLine = Utils.FormatHexString(binLine.toUpperCase(), 4);
129         idxStr = Utils.FormatBinString((i + ""), 4, true);
130
131         nret.add("@DCHW #" + hex + "\t\t;index: " + idxStr + "\\
132 taddress: " + binLine);
133         tCount++;
134     }
135
136     nret.add("");
137     asmFileAdjTypes.put(s, idxs[0]);
138     asmFileAdj.put(s, nret);
139     AddPrefix(nret, whiteSpace);
140     asmFileAdjNames.put(s, fileName);
141 } else {
142     throw new ExceptionUnsupportedAssemblyFileType("Cannot load
binary files without file extension .bin or .raw for file name, " +
fileName);
143 }
144 } ...

```

The PreProcessorThumb class' main method listing part 3, showing the include binary preprocessor directive.

The include binary preprocessor directive section starts on line 63 with a check to make sure the determined directive is correct. Note that this section of code supports both the `$INCBIN` and the `$FLPINCBIN` preprocessor directives. These directives are very similar. The only difference is that in one case we flip the encoding of the binary data from big-endian to little-endian. This allows us to control the encoding of the binary files on the fly.

The code on lines 64 – 69 should look familiar to you. The include binary preprocessor directive requires a file argument in the same way the include assembly directive does. As such, the first few lines of code in this code block are the same. Next up, on line 71 we check the file extension of the specified data file to make sure it's the proper type. The preprocessor supports bin, raw, and dat file extensions with pure binary data as the contents of file. On lines 72 – 75 we prepare some local variables for use with this directive.

The specified binary data file is loaded on line 72, in the `incBin` byte array. The `numBytes`, `numHalfWords`, and `padding` variables are set on lines 73 – 75. The snippet of code on lines 77 – 79 is there to check if the included binary file exceeds the maximum allowed size. If so, we throw an exception if not, we check the length of data to see if it's half-word aligned. Remember we're working with a 16-bit wide instruction set.

If we determine that there's an odd number of bytes, then we turn padding on because we want to ensure half-word alignment by adding a zero byte to the end of the included binary data. Line 85 introduces a local variable, `shorts`, that's designed to hold an array of short integer values, 16-bit, that will be inserted into the assembly program as data directives.

Take a moment to think about the power of this preprocessor directive and in fact it is very powerful. I'll support my opinion with an argument. The emulator we're going to be running our test programs on support both Thumb-1 and 32-bit ARM instructions. The emulator actually starts in 32-bit mode, we use a special branch instruction to jump to a specific line of code and tell the CPU to switch to Thumb-1 instruction set mode.

How can we do this if our compiler only supports 16-bit Thumb-1 instructions you ask. Well, that's where the binary include directive comes in

handy. We captured the binary code that performs the jump and saved it as raw binary data. Then we include it, in binary form, at the start of the test programs. The code ensures the instructions at a certain address in the program are handled in Thumb-1 mode.

You'll get more exposure to this topic when we talk about the test programs. For now, let's get back to the code review. On line 86 a call is made to convert the `incBin` byte array to an array of shorts. In the snippet of code on lines 87 to 95 there are a number of local variables declared and prepared. Let's take a closer look. Line 87 instantiates a new `List` of strings named `nret`. This is a local variable that is designed to hold the new lines of assembly source code that are generated by processing the specified data file.

The next two entries in this snippet are the `hex` and `tIncBin` variables. These are used in the conversion of data from binary to hex before being converted into a data directive. Using hex values is easier to read in assembly source code so we need to convert the binary values and these variables help with that process. The next four variables are for loop control, counting, and temporary use during binary conversion tasks. Lines 94 and 95 deserve some attention. We're using the data structure that contains the new assembly code to hold header information about the included file and a spacing line at the footer to separate the injected code from the next line of original assembly source code.

The for loop on lines 97 – 110 is where the real work for this directive is done. The loop iterates over the `shorts` array adding data directives to the `nret` list variable. A hex representation of the current short value is prepped on lines 98 and 99. An important if statement resides on lines 100 – 102. Recall that this preprocessor directive accepts both a direct binary inclusion and a flipped binary inclusion. We can quickly and easily handle this case with a call to the `EndianFlipHex` method.

The hex value is formatted on line 103 and a binary representation of the new line number, that the generated assembly code will reside on, is calculated on line 104, and subsequently formatted on line 105. Next, on line 106 we prepare an integer representation of the new line of source code. On line 108 the new data directive is added to a list for future injection to the original source code. Notice that a fair amount of information about the binary

data being written is included as a comment at the end of the line. This can help greatly with debugging any issues that may come up in an assembly program.

The block of code on lines 114 – 133 is very similar to the code we've just reviewed. This code executes in the case that padding is needed when converting the binary data to assembler data directives. Take a moment to look it over and make sure that you understand it before moving on. On line 135 we add some spacing to the end of the binary data inserted and then on lines 136 – 139 we register the new lines of source code, in the same way we've done previously, so that they can be written out with the new version of the source code file.

That brings us to the conclusion of this particular preprocessor directive review. We have just one more directive left to review before we take a look at the code responsible for writing out all the changes we've logged. The last two directives we'll look at are the string and flip string directives. These are used to convert string text to 16-bit data directive entries much like we've just seen with the include binary preprocessor directives.

*Listing 14-9. PreProcessorThumb.java Main Method Details 4*

```
144 } else if (idxs[0] == PPD_STRING_IDX || idxs[0] == PPD_FLIPSTRING_IDX)
{
145     sIdx = idxs[1];
146     tmp = s.substring(s.indexOf("|") + 1, s.lastIndexOf("|")).trim();
147
148     fileName = "String replacement: " + tmp;
149     tmp += ((char) PPD_EOL_BYT
E) + "";
150     char[] crs = tmp.toCharArray();
151     int len = crs.length;
152     if (len % 2 != 0) {
153         paddingOn = true;
154         len++;
155     }
156
157     byte[] binCrs = new byte[len];
158     for (int i = 0; i < len; i++) {
159         if (i < crs.length) {
160             binCrs[i] = ((byte) crs[i]);
161             Logger.wrl("Char: " + tmp.charAt(i) + " Val: " + crs[i]);
```

```

162     } else {
163         binCrs[i] = PPD_EOL_BYTE;
164     }
165 }
166
167 numBytes = len;
168 numHalfWords = numBytes / 2;
169 if (numBytes > MAX_STRING_LEN) {
170     throw new ExceptionUnsupportedStringLength("Cannot load an in-
line string with length greater than " + MAX_STRING_LEN + " characters");
171 }
172
173 short[] shorts = new short[numHalfWords];
174 ByteBuffer.wrap(binCrs).asShortBuffer().get(shorts);
175 List<String> nret = new ArrayList<>();
176 String hex = null;
177 int tCount = 0;
178 String binLine = null;
179 int i = 0;
180 String idxStr = null;
181 nret.add(";Found string byte count: " + numBytes + ", word count: "
+ numHalfWords + ", and padding on: " + paddingOn);
182 nret.add("");
183
184 for (i = 0; i < shorts.length; i++) {
185     hex = Integer.toHexString(shorts[i] & 0xffff);
186
187     if (idxs[0] == PPD_FLIPSTRING_IDX) {
188         hex = UtilsEndianFlipHex(hex);
189     }
190
191     hex = UtilsFormatHexString(hex.toUpperCase(), 4);
192     binLine = Integer.toHexString((tCount * 2) & 0xffff);
193     binLine = UtilsFormatHexString(binLine.toUpperCase(), 4);
194     idxStr = UtilsFormatBinString((i * 2 + ""), 4, true);
195
196     if ((i * 2 + 1) < binCrs.length) {
197         nret.add("@DCHW #" + hex + "\t\t;index: " + idxStr + "\t" +
(char) binCrs[i * 2] + " " + (char) binCrs[i * 2 + 1]);
198     } else {
199         nret.add("@DCHW #" + hex + "\t\t;index: " + idxStr + "\t" +
(char) binCrs[i * 2]);
200     }

```

```
201     tCount++;
202 }
203
204 nret.add("");
205 asmFileAdj.put(s, nret);
206 AddPrefix(nret, whiteSpace);
207 asmFileAdjNames.put(s, fileName);
208 }
```

The PreProcessorThumb class' main method listing part 4, showing the include string preprocessor directive.

Let's take a moment to think about the last set of directives we have to review. It's almost the same as including binary data, isn't it? All we're doing is converting the string to binary data and registering where in the original file the new lines of assembly source code need to be injected. We'll see that the code is very similar to the previously reviewed directive. Let's take a look!

On line 145 we record the position in the original line of source code where the directive was found and on line 146, we extract the string that we're going to be processing in much the same way we extracted the files that were specified for the include assembly and include binary directives. Next up, on lines 148 – 155, local variables are prepared and the need for padding is detected. Note that we also have a special byte that is used to indicate the end of a string. This combined with the padding can increase the binary representation of the string by two bytes.

The provided string's characters are converted into bytes on lines 157 – 165. This code is straightforward. Take a moment to look it over and make sure you understand it before moving on. We calculate the number of bytes and half-words needed to represent the string and check it does not exceed the maximum supported string length on lines 167 – 171. We instantiate a byte array, `shorts`, in much the same way we did for the previous directive. The `shorts` array is initialized and a few, familiar, local variables are defined on lines 175 – 180.

Similar to the include binary file directive we include a header and spacing at the beginning of the newly generated assembly source code. You

can see this in action on lines 181 – 182. The conversion of the string data to assembler data directives takes place in the for loop on lines 184 – 202. This code is very similar to what we've seen prior in this chapter. It's a bit redundant to cover it again. Take a look at it and make sure you understand the conversions and formatting being used.

Also, note that the include string directive appends extra information, debugging etc., at the end of each generated line of assembly source code, similar to the previous preprocessor directives we've just reviewed. Some spacing is appended to the new block of source code on line 204 and the lines are prefixed with the proper spacing, so they match surrounding original source code better. Lastly, the changes we've created are registered in the `asmFileAdj` and `asmFileAdjNames` fields, as we've seen with the prior preprocessor directives we've covered in this chapter.

That brings us to the last section of this method review. Where we'll see how all the registered changes are processed and used to create a new version of the assembly source code with all preprocessor directives properly and cleanly handled. Before we get into this next bit there is a small but important snippet of code on lines 214 – 219. I didn't put it into the listing because of how much code there was to review but you can take a look at it for yourself.

This is a subtle snippet of code, can you tell what it's doing? In short, it is replacing all the innocuous no-operation preprocessor directives with the generated assembly source code. The reason we can do this with such ease is that the replacement is 1 to 1, no new lines of assembly source need to be injected. We simply replace the original line for the generated one and we're done.

#### *Listing 14-10. PreProcessorThumb.java Main Method Details 5*

```
221 int idxTmp = -1;
222 int rowCountOrig = -1;
223 int rowCountNew = -1;
224 int type = -1;
225 fileName = null;
226 for (String key : asmFileAdj.keySet()) {
227     rowCountOrig = ret.size();
228     fileName = asmFileAdjNames.get(key);
```

```

229     idxTmp = ret.indexOf(key);
230     ret.addAll(idxTmp + 1, asmFileAdj.get(key));
231     rowCountNew = ret.size();
232
233     if (asmFileAdjTypes.containsKey(key) == true) {
234         type = asmFileAdjTypes.get(key);
235     } else {
236         type = -1;
237     }
238
239     if (type == PPD_INCBIN_IDX || type == PPD_INCASM_IDX) {
240         Logger.wrl("PreProcessorThumb: Adding inline assembly at line "
241 + (idxTmp + 1) + " from file, " + fileName);
242         Logger.wrl("PreProcessorThumb: Assembly file row count before "
243 include, " + rowCountOrig + ", and after include, " + rowCountNew + ".");
244     } else {
245         Logger.wrl("PreProcessorThumb: Adding inline assembly at line "
246 + (idxTmp + 1) + " from string, " + fileName);
247         Logger.wrl("PreProcessorThumb: Assembly file row count before "
248 include, " + rowCountOrig + ", and after include, " + rowCountNew + ".");
249     }
250 }
251
252 FileUnloader.writeFileList(Paths.get(rootOutputDir,
253 OUTPUT_FILE_NAME).toString(), ret);
254 return ret;

```

The PreProcessorThumb class' main method listing part 5, showing the generation of the modified assembly source code file.

Let's wrap up this method review and see how all the changes we've registered are used to modify the original assembly source code. On lines 221 – 225 we prepare some local variables for the task at hand, `idxTmp`, `rowCountOrig`, `rowCountNew`, and `type`. We'll also need the `fileName` variable, so we make sure to clear it here. On line 226 we iterate over the keys in the `asmFileAdj` map. The size of the original assembly source code is set on line 227 while the file name associated with this directive is pulled and stored in the `fileName` variable on line 228.

Note that the `rowCountOrig` variable is updated after each iteration of the loop to show the change in row count as we process the logged adjustments. Also note that all adjustments entries logged use the same key, the original source code line that contained the preprocessor directive with the specified file, that is why we can iterate over the `asmFileAdj` map's key set and use the same key to pull out the associated file name. It is also why we are driving the process by the entries in the `asmFileAdj` field.

There is a limitation to this implementation, can you spot it? If we use the same include preprocessor directive with the same file associated we'll get an error using it as a key in the Maps we use to log the adjustments. This should be fine for our needs which are more educational but keep it in mind. You can always copy the file and name it something slightly different, add a -1 or -2 to the name, to get around the limitation.

The position of the current line of source we're looking at is stored in the `idxTmp` variable on line 229. Line 230 is deceptively important. It handles injecting the newly generated source code into the correct location in the original file. Now can you see why it made sense to use the assembly source and not a line number to track changes? We'd have to recalculate every line of every change after the prior change is implemented to take into account the line number changes.

Line 231, the `rowCountNew` variable is updated to reflect the changes to the original source code, contained in the `ret` variable. The last bit of code in the method is used to determine the original type of preprocessor directive and include that information in the method's logging output. Finally, on line 248, the new assembly source code is written to a file and returned from the method. That brings us to the conclusion of the main method details review. In the next section we'll take a look at how the class is used.

## Demonstration: PreProcessorThumb.java

To demonstrate the class in action we'll take a look at a familiar line of code from the project's static main method.

*Listing 14-11. PreProcessorThumb.java Demonstration*

File: GenAsm

```

Method: main
Snippet:
01 if (ASM_SET != null) {
02     if (ASM_ASSEMBLER != null && ASM_PREPROCESSOR != null &&
ASM_LINKER != null) {
03         List<String> fileData =
ASM_PREPROCESSOR.RunPreProcessor(ASM_ASSEMBLY_SOURCE_FILE,
ASM_ROOT_OUTPUT_DIR, null);
04         ASM_ASSEMBLER.RunAssembler(ASM_SET, ASM_ASSEMBLY_SOURCE_FILE,
fileData, ASM_ROOT_OUTPUT_DIR, null, null, ASM_VERBOSE,
ASM_QUELL_FILE_OUTPUT);
05         ASM_LINKER.RunLinker(ASM_ASSEMBLER, ASM_ASSEMBLY_SOURCE_FILE,
ASM_ROOT_OUTPUT_DIR, null, ASM_VERBOSE, ASM_QUELL_FILE_OUTPUT);
06     } else {
07         Logger.wrl("GenAsm: Main: Error: could not find properly loaded
pre-processor, assembler, or linked");
08     }
09 } else {
10     Logger.wrl("GenAsm: Main: Error: could not find assembler set named
" + ASM_TARGET_SET);
11 }

```

A sample listing showing the PreProcessorThumb class in action.

The example shown in the previous listing is from the project's static main entry point. After a few sanity checks to make sure our variables are defined we quickly call the `RunPreProcessor` method, line 3, and provide the necessary arguments. Notice that the results of the method call are stored in the `fileData` local variable and that this variable is then provided to the assembler via the `RunAssembler` call.

Before I close out this section, I wanted to further this demonstration with some snippets from the assembly test program, `TEST_N_AsmChecks`. Take a moment to look at the following snippet of assembly source code found in the following source code file.

`.\\cfg\\THUMB\\TESTS\\TEST_N_AsmChecks\\genasm_source.txt`

Take a look at these lines of code at the top of the file:

```
;+++++
;$INCASM | .\cfg\THUMB\INCBIN\test_asm_incasn.txt|
;+++++
```

Now, let's look at the output from the preprocessor for this test program. The output file should already be included in the output folder but if it's not you may need to run this program to generate it. In either case I'll list the operative lines of code here:

```
.cfg\THUMB\OUTPUT\TEST_N_AsmChecks\
output_pre_processed_assembly.txt
```

The few lines of code at the top of the file now look like this:

```
;+++++
;$INCASM | .\cfg\THUMB\INCBIN\test_asm_incasn.txt|
;Found file with line count: 4, byte count: 86
```

```
LDR      R3, [PC, #844]
STR      R3, [R2, R6]
LDRB     R2, [R0, R7]
STRH     R4, [R3, R0]
```

```
;+++++
```

The preprocessor took care of injecting the new source code into the original file and adding some annotations that help us understand where the assembly source code came from. Notice that the byte count is 86. You might be saying that there's only 16 bytes of assembly there. Remember this preprocessor directive injects text, 86 characters of it, that is, expected to be, valid ARM Thumb-1 assembly source code. That's really all that I wanted to cover with regard to using this class. Notice how simple it is to use the class.

It's very powerful to encapsulate complex code, like we've seen with the preprocessor, in simple easy to use classes that do not take a lot of setup and configuration.

## Chapter Conclusion

In this chapter we took a look at one of the key aspects of the GenAsm project, and really any assembler for that matter, the preprocessor. Let's review some of the main topics we've touched upon during this class review.

- **Interfaces:** This chapter really demonstrates the benefits of using interfaces in your software design. It would be very easy to add a new preprocessor into the system and that flexibility is by design and carried forward through the use of interfaces.
- **The \$NOP Directive:** We took a look at how the innocuous instruction directive is handled and saw how the preprocessor changes are registered for future handling as opposed to making changes to the source in-line with the directive scanning. We also saw how \$NOP directives are replaced one-to-one and are handled separately before the include preprocessor directives.
- **The \$INCASM Directive:** The include assembly directive demonstrated to us how to inject lines of assembly into an existing program. This directive can be used to encapsulate and centralize some commonly used assembly code that's shared by multiple programs.
- **The \$INCBIN, \$FLPINCBIN Directives:** We got to take another look at a way to inject assembly binary code into our assembly program. In this case we reviewed the code responsible for reading a raw binary file and converting it to data directives that are injected into the original source code.
- **The \$STRING, \$FLIPSTRING Directives:** Last but not least we took a look at yet another way to inject binary data into our program. In this case we reviewed the string directives that are responsible for converting a specified string to a series of data directives.

This chapter was rather rigorous. I'd like to say that it'll get easier from here on out, but the truth is we have some serious code to review. That being said you can review things at your own pace, don't feel pressure to understand everything in its entirety the first time through. In the next chapter we'll take a look at the next step in the assembly process, the lexer.

# **Chapter 15: The Lexer**

The next step in our assembler exploration journey is the lexer. The lexer is used to parse assembly source code and separate it into artifacts. Just what sort of separation you're doing and what kind of artifacts you're capturing is a function of the assembly source code you are trying to parse. That being said, in our particular circumstance we're interested in all pieces of text sans whitespace.

In this chapter we'll take a look at the Thumb-1 implementation of the lexer interface. This class is not concise, so we'll employ our more robust class review protocol. I'll list the sections we'll visit here:

- Static/Constants/Read-Only Class Members
- Class Fields
- Pertinent Method Outline/Class Headers
- Support Method Details
- Main Method Details
- Demonstration

There are no pertinent enumerations to speak of. Although this class is not short it's not as long as the preprocessor class we've just reviewed.

## Class Review: `LexerThumb.java`

The `LexerThumb` class takes us one step closer to the core assembler work. In this class we'll scan through the source code and separate the fragments of text known as artifacts by parsing it based on some basic rules that apply to the ARM Thumb-1 instruction set. We know what our assembly source code is going to look like so we're not going to define a complex system to describe it, we're just going to implement it directly in the `LineLexerize` method. Let's refresh our memory as to the structure of the source code we're parsing.

*Listing 15-1. GenAsm Assembly Source Code Example*

```
ASR      R0, R1, #31
ADD      R0, R1, R2      ;5.2
ADD      R0, R1, #7
SUB     R0, R1, R2
SUB     R0, R1, #7
```

A sample listing of the GenAsm assembler's implementation of the Thumb-1 instruction set.

As you can see the structure of assembly code is relatively simple. There are few complex structures and none that extend past one line. This greatly simplifies our task of converting the text into binary form. The first section we'll look into is the static class fields section.

## Static/Constants/Read-Only Class Members: `LexerThumb.java`

The `LexerThumb` class has a number of static class fields that we'll take a look at. They are mainly used to describe the special characters we need to be aware of while parsing the assembly source code. Take a look at them and imagine how they might be used to parse our assembly source code.

## *Listing 15-2. LexerThumb.java Static/Constants/Read-Only Class Members*

```
public static char[] CHAR_SEPARATORS = { ',', ' ', '\t', ';' };
public static char[] CHAR_STICKY_SEPARATORS = { } ;
public static char[] CHAR_NEW_ARTIFACT_SEPARATORS = { ';' } ;
public static char[] CHAR_GROUP_START = { '[', '{' } ;
public static char[] CHAR_GROUP_STOP = { ']', '}' } ;
public static char[] CHAR_WHITE_SPACE = { ' ', '\t' } ;
```

The static class fields of the LexerThumb class.

The first static class field entry is the `CHAR_SEPARATORS` array. This array stores all of the characters that indicate a separation between artifacts. Most of them are to be expected based on the general syntax of assembly source code. Some of them are specific to this implementation. The semicolon is there not to indicate the end of a line of assembly source code. It's used to indicate a comment has started, and in this case the semicolon would be surrounded by white-space like so, ‘ ; Comment starts here’. The next character specification is empty, this field lets you specify which characters are “sticky.” That is to say they are kept attached to the artifact after extraction. We've set things up so that we don't need any sticky characters.

The next field, `CHAR_NEW_ARTIFACT_SEPARATORS`, ties into the semicolon separator character we just finished discussing. In this case, the semicolon is immediately followed by text. And in this case we'd like to keep the text attached to the semicolon and create one new artifact as opposed to creating one semicolon separator artifact and one that contains the subsequent text.

The next two fields, `CHAR_GROUP_START` and `CHAR_GROUP_STOP`, are used to indicate the group and list start and stop characters. These character come into play with certain opcodes that have a group or list of arguments associated with them. Lastly, the whitespace characters that are ignored by the parse are listed in the `CHAR_WHITE_SPACE` static class field. In the next section we'll take a look at the remaining class fields.

## Class Fields: LexerThumb.java

There is only one class field remaining for us to review.

*Listing 15-3. LexerThumb.java Class Fields*

```
public String obj_name = "LexerThumb";
```

The LexerThumb class' only non-static class field.

The LexerThumb class has only one class field for us to look at. This field should be familiar, we've seen it before in both loader and holder class reviews. The `obj_name` field holds the name of the LexerThumb class for consistency and to aid in connecting a JSON representation of this class, if any, with the actual Java class.

## Pertinent Method Outline/Class Headers: LexerThumb.java

In this section we'll take a look at the LexerThumb class' pertinent methods, listed as follows.

*Listing 15-3. LexerThumb.java Pertinent Method Outline/Class Headers 1*

```
//Main Methods
public ArrayList<ArtifactLine> FileLexerize(List<String> file);
public ArtifactLine LineLexerize(String line, int lineNumber);

//Support Methods
private boolean Contains(char[] array, char subj);
```

A listing of the LexerThumb class' pertinent methods.

There are only a few methods for us to review, shown in the previous listing. The main methods are responsible for processing the file's contents

and the support methods are loosely considered to support them. We'll take a look at these methods in detail in just a bit. Next let's take a look at the class' declaration.

*Listing 15-4. LexerThumb.java Pertinent Method Outline/Class Headers 2*

```
package net.middlemind.GenAsm.Lexers.Thumb;

import java.util.ArrayList;
import java.util.List;
import net.middlemind.GenAsm.Lexers.Artifact;
import net.middlemind.GenAsm.Lexers.ArtifactLine;
import net.middlemind.GenAsm.Lexers.Lexer;
import net.middlemind.GenAsm.Utils;

public class LexerThumb implements Lexer { }
```

The LexerThumb class' declaration.

Take a moment to look over the class declaration shown in the previous listing. Pay attention to the imports and interface implemented by the class. There's not much more to cover in this section so let's move on to a review of the class' support method.

## Support Method Details: LexerThumb.java

The LexerThumb class has only one support method for us to review, the Contains method.

*Listing 15-5. LexerThumb.java Support Method Details*

```
1 private boolean Contains(char[] array, char subj) {
2     for (char c : array) {
3         if (c == subj) {
4             return true;
5         }
6     }
7     return false;
8 }
```

The `LexerThumb` class' Contains method.

This method is simple in nature. It searches a provided array of characters for an instance of a specific character, also provided as an argument. That's all we have to say about this method. Short and sweet. Let's move on to the review of the class' main methods.

## Main Method Details: `LexerThumb.java`

The `LexerThumb` class has two main methods for us to review. Both are defined by the `Lexer` interface and are required to be implemented by the `LexerThumb` class. We'll start with the file level method, listed here.

*Listing 15-6. `LexerThumb.java` Main Method Details 1*

```
1 public ArrayList<ArtifactLine> FileLexerize(List<String> file) {  
2     ArrayList<ArtifactLine> ret = new ArrayList<>();  
3     int count = 0;  
4     for (String s : file) {  
5         ret.add(LineLexerize(s, count));  
6         count++;  
7     }  
8     return ret;  
9 }
```

The `LexerThumb` class' implementation of the `FileLexerize` method.

The `FileLexerize` method is designed to work with the `List` of strings we use to represent an assembly source code file. In this case it is the method's only argument. The method simply iterates over the contents of the file and runs the `LineLexerize` method on each line found, passing in the line number it was found on along with the source code of the line, line 5. Notice that this method returns an `ArrayList` of `ArtifactLine` instances. We'll take a look at an output file created from the lexer's results as part of the demonstration section, after we complete the main method review.

You'll really get to see, first-hand, what constitutes an artifact. We can see that this class converts the string representation of the source code into an `ArtifactLine` instance. In the next step of the assembly process, the tokenizer converts an `ArtifactLine` into a `TokenLine`. We'll get to that part in the next chapter. For now, let's direct our attention to the subsequent method listing.

***Listing 15-7. LexerThumb.java Main Method Details 2***

```
001 public ArtifactLine LineLexerize(String line, int lineNumber) {  
002     ArtifactLine ret = new ArtifactLine();  
003     ret.source = line;  
004     ret.lineNum = lineNumber;  
005  
006     if (Utils.IsEmpty(line)) {  
007         ret.payload = new ArrayList<>();  
008         ret.sourceLen = 0;  
009         return ret;  
010     } else {  
011         ret.sourceLen = line.length();  
012         ret.payload = new ArrayList<>();  
013  
014         char[] chars = line.toCharArray();  
015         boolean inArtifact = false;  
016         Artifact artifact = null;  
017         int i = 0;  
018         int count = 0;  
019  
020         for (; i < chars.length; i++) { ...
```

The `LexerThumb` class' implementation of the `LineLexerize` method, part 1.

This method is somewhat lengthy, so I'll separate the code review into blocks. In the first block, listed previously, the field and variable preparation is done before any work on the line is done, lines 2 – 18. On lines 2 – 4 we prepare the return object, and `ArtifactLine` instance, and set its `source` and `lineNum` fields. If the line we're processing is empty, line 6, we proceed to create an empty `ArtifactLine` and return it on lines 7 – 9. If not, we

prepare to parse the line of source code by initializing the `sourceLen` and `payload` fields of the `return` object, lines 11 – 12.

The next few lines, 14 – 18, prepare a few local variables for use by the main processing loop. The first is the `chars` variable. This variable is set by converting the line of source code to an array of characters, line 14. The `inArtifact` Boolean is a flag that tracks if we are currently building an existing `Artifact` or if we're starting to build a new one. The `artifact` field is used to hold the `Artifact` we're currently working on. Recall that an `ArtifactLine`, much like a `TokenLine`, is comprised of a list of `Artifact` objects and some associated data like the `source` and `sourceLen` fields. The next two fields, `i` and `count`, are loop control variables. Iteration begins on line 20 at the end of the listing.

*Listing 15-8. LexerThumb.java Main Method Details 3*

```
021 if (inArtifact == true) {  
022     if (Contains(CHAR_SEPARATORS, chars[i])) {  
023         inArtifact = false;  
024         artifact.posStop = (i - 1);  
025         artifact.len = (i - artifact.posStart);  
026         artifact.index = count;  
027  
028         if (Contains(CHAR_STICKY_SEPARATORS, chars[i])) {  
029             artifact.payload += chars[i];  
030             artifact.posStop++;  
031             artifact.len++;  
032         }  
033  
034         ret.payload.add(artifact);  
035         count++;  
036         artifact = null;  
037  
038         if (Contains(CHAR_NEW_ARTIFACT_SEPARATORS, chars[i])) {  
039             artifact = new Artifact();  
040             artifact.lineNum = lineNumber;  
041             artifact.posStart = i;  
042             artifact.posStop = i;  
043             artifact.len = 1;  
044             artifact.index = count;  
045             artifact.payload = "" + chars[i];
```

```

046         ret.payload.add(artifact);
047         count++;
048         artifact = null;
049     }
050
051 } else {
052     if (Contains(CHAR_GROUP_START, chars[i])) {
053         inArtifact = false;
054         artifact.posStop = (i - 1);
055         artifact.len = (i - artifact.posStart);
056         artifact.index = count;
057
058         if (Contains(CHAR_STICKY_SEPARATORS, chars[i])) {
059             artifact.payload += chars[i];
060             artifact.posStop++;
061             artifact.len++;
062         }
063
064         ret.payload.add(artifact);
065         count++;
066         artifact = null;
067
068         artifact = new Artifact();
069         artifact.lineNum = lineNumber;
070         artifact.posStart = i;
071         artifact.posStop = i;
072         artifact.len = 1;
073         artifact.index = count;
074         artifact.payload = "" + chars[i];
075         ret.payload.add(artifact);
076         count++;
077         artifact = null;
078
079     } else if (Contains(CHAR_GROUP_STOP, chars[i])) {
080         inArtifact = false;
081         artifact.posStop = (i - 1);
082         artifact.len = (i - artifact.posStart);
083         artifact.index = count;
084
085         if (Contains(CHAR_STICKY_SEPARATORS, chars[i])) {
086             artifact.payload += chars[i];
087             artifact.posStop++;
088             artifact.len++;

```

```

089         }
090
091         ret.payload.add(artifact);
092         count++;
093         artifact = null;
094
095         artifact = new Artifact();
096         artifact.lineNum = lineNumber;
097         artifact.posStart = i;
098         artifact.posStop = i;
099         artifact.len = 1;
100         artifact.index = count;
101         artifact.payload = "" + chars[i];
102         ret.payload.add(artifact);
103         count++;
104         artifact = null;
105
106     } else {
107         artifact.payload += chars[i];
108
109     }
110 }
111 } else { ...

```

The LexerThumb class' LineLexerize method handling separation, group start, and group stop character sets when building an Artifact is already in progress. Part 2 of the method review.

The block of code shown in the previous listing handles processing the source line's characters when we're already building an `Artifact`, line 21. The first case we look at, line 22, is if the current character is one of the separator characters. If we encounter a separator character, we stop building the current artifact and set its `posStop` position, `len`, and `index` fields, lines 23 – 26. Remember this code runs when we've already started building an artifact. The `posStop` field is set to the value, `(i - 1)`, because we don't include the separator. We can set the `len` field to the value, `(i - artifact.posStart)`, because the positioning is zero based.

Next, on lines 28 – 32, we check to see if the current character we’re processing is a sticky character. In which case, we append it back to the artifact’s payload field and increment the posStop and len fields. On lines 34 – 36, the artifact is added to the artifact line and the local variable is reset. We also increment the count variable which keeps track of the number of Artifact instances on this ArtifactLine.

The snippet of code on lines 38 – 49 is responsible for handling a new artifact separator character. The only character that currently holds this property is the comment character, ‘;’, which can appear at any point in the source code line. As such, we have to create a new Artifact to indicate that we’ve encountered the character.

Note that the semicolon is also a separator character. Can you see how the two character separators and new artifact separators work together to handle the specific case of the comment character? It is used to create a separation between artifacts but is then otherwise lost. Making it a sticky character would not fix the problem because then it would be appended to the source field of the previous artifact. The solution is to detect it after the separation character is handled and create a new artifact instance to mark it if need be. The code on lines 39 – 48 takes care of creating a new Artifact to mark the occurrence of the character. It then appends it to the ArtifactLine’s payload, and resets the appropriate local variables.

The next big block of code, lines 52 – 106, handles the start and stop characters for groups and lists. The code is very similar to the listings we’ve just reviewed. The current artifact being built is completed and a new artifact to mark the start or stop character is created and appended to the return payload, lines 75 and 102. Take a moment to look over this code and make sure you understand it before moving on.

The last few lines of code in the previous listing handles the case if the inArtifact Boolean is true and the current character does not match any of the character checks we’ve performed thus far in the method. In this case we simply append the character to the current artifact’s payload field, line 107. We’ve seen how the artifacts are built up over loop iterations and ended by certain special characters. In the next listing we’ll see how the artifacts are initialized.

*Listing 15-9. LexerThumb.java Main Method Details 4*

```
112 if (Contains(CHAR_WHITE_SPACE, chars[i])) {  
113     //ignore whitespace  
114     continue;  
115  
116 } else if (Contains(CHAR_GROUP_START, chars[i])) {  
117     artifact = new Artifact();  
118     artifact.lineNum = lineNum;  
119     artifact.posStart = i;  
120     artifact.posStop = i;  
121     artifact.len = 1;  
122     artifact.index = count;  
123     artifact.payload = "" + chars[i];  
124     ret.payload.add(artifact);  
125     count++;  
126     artifact = null;  
127  
128 } else if (Contains(CHAR_GROUP_STOP, chars[i])) {  
129     artifact = new Artifact();  
130     artifact.lineNum = lineNum;  
131     artifact.posStart = i;  
132     artifact.posStop = i;  
133     artifact.len = 1;  
134     artifact.index = count;  
135     artifact.payload = "" + chars[i];  
136     ret.payload.add(artifact);  
137     count++;  
138     artifact = null;  
139  
140 } else {  
141     inArtifact = true;  
142     artifact = new Artifact();  
143     artifact.lineNum = lineNum;  
144     artifact.posStart = i;  
145     artifact.payload = "" + chars[i];  
146  
147 }
```

A listing of the Artifact starting conditions. This code initializes the artifact instance that will be built up by the loop. Part 3 of the method review.

The code shown in the previous listing demonstrates all the conditions where we start to create a new `Artifact` to be built up over subsequent loop iterations. In the first case, however, lines 112 – 115, detected white-space is ignored. The code on lines 116 – 140 is very direct. If we encounter a group start or stop character we create a new `Artifact` instance, add it to the line's `payload` and then manage some loop variables, 125, 126 and 137, 138. This is very similar to the way we handled the group start and stop character in the previous code block.

Note that group start and stop characters get their own artifact, they are not combined with the artifacts in the group. The last few lines of code in the listing, lines 141 – 145, create a new `Artifact` based on the current character. Note that at this point in the code the current character isn't a special character.

#### *[Listing 15-10. LexerThumb.java Main Method Details 5](#)*

```
151 if (artifact != null) {  
152     artifact.posStop = (i - 1);  
153     artifact.len = (i - artifact.posStart);  
154     artifact.index = count;  
155     ret.payload.add(artifact);  
156 }  
157  
158 return ret;
```

The last few lines of the `LineLexerize` method. Part 4 of the method review.

The remaining lines of code in the method are there to make sure any `Artifact`'s that we're being worked on in the loop when it ends are not lost. We make sure to process them and add them to the return line's `payload`. In the next section we'll take a look at a demonstration of this class, and we'll take a moment to look at the output file generated at this point in the assembly process.

## Demonstration: LexerThumb.java

For the demonstration of the `LexerThumb` class we'll look at the following support method from the `AssemblerThumb` class.

*Listing 15-11. `LexerThumb.java` Demonstration 1*

File: `AssemblerThumb.java`

Method: `LexerizeAssemblySource`

Snippet:

```
01 public void LexerizeAssemblySource(int step) throws IOException {
02     if (eventHandler != null) {
03         eventHandler.LexerizeAssemblySourcePre(step, this);
04     }
05
06     Logger.wrl("AssemblerThumb: LoadAndLexAssemblySource: Lexerize
assembly source file");
07     LexerThumb lex = new LexerThumb();
08     asmDataLexed = lex.FileLexerize(asmDataSource);
09
10    if (eventHandler != null) {
11        eventHandler.LexerizeAssemblySourcePost(step, this);
12    }
13 }
```

This listing shows the `LexerizeAssemblySource` support method and its use of the `LexerThumb` class.

The previously shown method is responsible for running the lexer on the assembly source code loaded by the `AssemblerThumb` class as part of the assembly process. On lines 2 – 4 we provide a chance to customize the process with access to callback methods. The lexer is run on lines 7 – 8, and a closing callback method is called, if defined, on lines 10 – 12. That's really all there is to it. The implementation of the lexer is simple and pain free. Before we move on from this section, I want us to look at the data involved with this process. We'll take a look at a line of source code followed by a JSON representation of the lexer's results.

*Listing 15-12. LexerThumb.java Demonstration 2*

```
1      test      @EQU    2
```

A sample line of Thumb-1 assembly source code.

A simple line of assembly source code is shown in the previous listing. The results of the `LexerThumb` class' operation on this line of source code is as follows. Notice the use of the `obj_name` field? It provides some important information about what object is being written. We've seen it in a lot of class reviews, now we're seeing it in action.

*Listing 15-13. LexerThumb.java Demonstration 3*

```
{
  "obj_name": "ArtifactLine",
  "lineNum": 5,
  "sourceLen": 25,
  "source": "      test      @EQU    2",
  "payload": [
    {
      "obj_name": "Artifact",
      "posStart": 4,
      "posStop": 7,
      "len": 4,
      "lineNum": 5,
      "index": 0,
      "payload": "test"
    },
    {
      "obj_name": "Artifact",
      "posStart": 16,
      "posStop": 19,
      "len": 4,
      "lineNum": 5,
      "index": 1,
      "payload": "@EQU"
    },
    {
      "obj_name": "Artifact",
```

```

    "posStart": 24,
    "posStop": 24,
    "len": 1,
    "lineNum": 5,
    "index": 2,
    "payload": "2"
}
]
}

```

A printout of an `ArtifactLine` object instance to a JSON object. The results of the `LexerThumb` class are written to the `output_lexer.json` data file.

Take a moment to look over the JSON representation of a lexerized line of source code. Make sure you understand it before moving on. Notice how only the non-whitespace text has been preserved after the lexer runs. This is perhaps a deceptively simple part of the assembly process. It's really important to have a good lexer that sets up your data structures well. In our case we have a nice clean representation of the source code to work with.

## Chapter Conclusion

In this chapter we took a look at the Thumb-1 implementation of the Lexer interface. The implementation class, `LexerThumb`, is used to prepare the preprocessed source code for analysis by the tokenizer. Let's summarize some of the key topics we've reviewed in this chapter.

- **LexerThumb Review:** We've completed a detailed review of the `LexerThumb` class covering all aspects of the class and its use in the GenAsm project.
- **The obj\_name Field in Action:** We got a chance to really see the `obj_name` field in action. This is a subtle but important point. Notice how the field brings a lot of meaning to the JSON object it resides in because it identifies the class that the JSON object was generated from.
- **Assembly Source to ArtifactLine JSON Object:** Lastly, we took a look at the results of the lexer operation. The GenAsm assembler is

designed as a learning tool and as such it outputs a lot of information about the assembly process. One such piece of information is the `output_lexed.json` data file.

We're slowly working our way to the core assembler code. In the next chapter we'll take a look at the implementation of the tokenizer that is used to convert `ArtifactLine` and `Arifact` objects to `TokenLine` and `Token` objects that have been identified and associated with an entry type.

# Chapter 16: The Tokenizer

The tokenizer's job is to convert the `Artifacts` generated by the lexer into `Tokens`. The difference between an `Artifact` and a `Token` is that an artifact is essentially just a string while a token is an artifact identified as being of a specific entry type. In other words, an identified string that has been assigned a type. Recall the entry types are defined as part of the instruction set in the `is_entry_types.json` data file. The entry type listings are used to identify an `Artifact` instance as being of a particular type. This is the basis for a `Token` in the GenAsm project.

In this chapter we'll take a look at the ARM Thumb-1 implementation of the `Tokenizer` interface. This class is somewhat lengthy, so we'll employ our more robust class review protocol. I'll list the sections we'll use here:

- Class Fields
- Pertinent Method Outline/Class Headers
- Main Method Details
- Demonstration

There are no static class fields, support methods, or pertinent enumerations to speak of, so we'll omit those sections. As we review the code try and imagine it being used to identify the different entry types available in the instruction set.

## Class Review: TokenerThumb.java

The `TokenerThumb` class is an important step in the assembly process. It allows us to move the comparison logic from text, `Artifact` based, to known, identified object comparisons, `Token` based. Let's jump into some code.

### Class Fields: TokenerThumb.java

The `TokenerThumb` class has only one field for us to review, listed as follows.

*Listing 16-1. TokenerThumb.java Class Fields*

```
public String obj_name = "TokenerThumb";
```

The `TokenerThumb` class' only class field.

The class' `obj_name` field is simple but important. As we have seen, the field provides information about the class associated with JSON object generated by the assembler. We've seen this field before during our review of the loader and holder classes. In the next section we'll take a look at the class' declaration.

### Pertinent Method Outline/Class Headers: TokenerThumb.java

In this section we'll take a look at the `TokenerThumb` class' pertinent methods and class declaration. Let's start things off with a quick look at the pertinent method outline.

*Listing 16-2. LexerThumb.java Pertinent Method Outline/Class Headers 1*

```
//Main Methods
public List<TokenLine> FileTokenize(List<ArtifactLine> file, JsonObject
entryTypes);

public TokenLine LineTokenize(ArtifactLine line, int lineNumber, JsonObject
entryTypes);
```

A listing of the TokenerThumb class' pertinent methods. There are only two main methods for us to review.

The `LexerThumb` class is an implementation of the `Lexer` interface that we reviewed in Chapter 15. There are two main methods defined by the class that we'll review here. The `FileTokenize` and `LineTokenize` methods. They are similar to those that were used by the `LexerThumb` class. In the next listing we'll take a look at the class declaration.

*Listing 16-3. LexerThumb.java Pertinent Method Outline/Class Headers 2*

```
package net.middlemind.GenAsm.Tokeners.Thumb;

import net.middlemind.GenAsm.Tokeners.Tokener;
import net.middlemind.GenAsm.Tokeners.TokenLine;
import net.middlemind.GenAsm.Tokeners.Token;
import net.middlemind.GenAsm.Lexers.ArtifactLine;
import net.middlemind.GenAsm.Lexers.Artifact;
import net.middlemind.GenAsm.Exceptions.Thumb.ExceptionMalformedRange;

import net.middlemind.GenAsm.Exceptions.Thumb.ExceptionNoTokenerFound;

import net.middlemind.GenAsm.JsonObjs.JsonObjTxtMatch;
import net.middlemind.GenAsm.JsonObjs.JsonObj;
import net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsEntryType;
import net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsEntryTypes;
import java.util.ArrayList;
import java.util.List;
import net.middlemind.GenAsm.Logger;
import net.middlemind.GenAsm.Utils;
```

```
public class TokenerThumb implements Tokener { }
```

A listing of the TokenerThumb class' declaration.

Take a moment to read over the class' declaration. Keep an eye out for any interesting import statements and any interfaces implemented by the class. In this case we're working with the `TokenerThumb` class which implements the `Tokener` interface we reviewed earlier in the text. In the next section we'll start the main method review.

## Main Method Details: TokenerThumb.java

The `TokenerThumb` class has two main methods for us to review. These methods are defined by the `Tokener` interface and implemented as part of the Thumb-1 instruction set implementation.

*Listing 16-4. TokenerThumb.java Main Method Details 1*

```
1 public List<TokenLine> FileTokenize(List<ArtifactLine> file, JsonObject
entryTypes) throws ExceptionNoTokenerFound {
2     int lineNumber = 0;
3     ArrayList<TokenLine> ret = new ArrayList<>();
4     for (ArtifactLine art : file) {
5         ret.add(LineTokenize(art, lineNumber, entryTypes));
6         lineNumber++;
7     }
8     return ret;
9 }
```

The TokenerThumb class' implementation of the FileTokenize method.

The `TokenerThumb` class implements a file processing method and a line processing method in much the same way that the `LexerThumb` class does. Notice that the `FileTokenize` method takes a lexered file, and an instance of the `JsonObjectIsEntryTypes` class, `entryTypes`, as arguments. On lines 2 – 3 we prepare some local variables. The `lineNum` variable, as you may have expected, keeps track of the current line number.

The next variable, `ret`, is an array of `TokenLine` objects and it is used to hold the results of processing each line in the `file` method argument. On line 5, we call the `LineTokenize` method and add the results to the `ret` variable. This object is returned at the end of the method, line 8. Next up, we'll take a look at the class' `LineTokenize` method.

*Listing 16-5. TokenerThumb.java Main Method Details 2*

```
001 public TokenLine LineTokenize(ArtifactLine line, int lineNumber, JsonObject
entryTypes) throws ExceptionNoTokenerFound {
002     JsonObjectIsEntryTypes types = (JsonObjectIsEntryTypes) entryTypes;
003     boolean found;
004     String payload = "";
005     Artifact current = null;
006     String compare = null;
007     String withResStarts = null;
008     String withResEnds = null;
009     String withResContains = null;
010     JsonObjectIsEntryType compareType = null;
011     int count = 0;
012     boolean inComment = false;
013     JsonObjectIsEntryType commentType = null;
014     boolean verbose = false;
015
016     TokenLine ret = new TokenLine();
017     ret.lineNumAbs = lineNumber;
018     ret.source = line;
019     ret.payloadLen = line.payload.size();
020     ret.payload = new ArrayList<>();
021
022     try {
023         for (Artifact art : line.payload) { ...
```

The `TokenerThumb` class' implementation of the `LineTokenize` method, part 1.

The `TokenerThumb` class' `LineTokenize` method is used to process each `ArtifactLine` of the `file` method argument. The method also takes an instance of a `JsonObject` class, `entryTypes`, as an argument. A number of

local variables are prepared on lines 2 – 14. The first one, line 2, sets the `types` local variable by casting the `entryTypes` method argument to an instance of the `JsonObjIsEntryTypes` class.

The `found` variable, line 3, is used to track the results of certain searches performed during the execution of this method. The `payload` variable is used to hold a copy of the current artifact's payload field, lines 4 – 5. The next four fields, lines 7 – 10, are used in the search for an entry type match. We'll get to know these variables better when we cover the method's core code. The `count` variable is used to track the number of `Artifact` objects processed.

The way the `LineTokenize` method is structured we iterate over each `Artifact` in the line. For each artifact we iterate over the instruction set's entry types and for each entry type we perform the following comparisons:

- Starts With String
- Ends With String
- Contains String
- Must Contain String
- Must Not Contain String

The local variable `inComment` is used to indicate if we're in a comment. This is a special case where the string matching no longer makes sense. We could potentially have commented assembly source code and we can't process it like active assembly source code. To this end we track if we have encountered a comment during the processing of the current `ArtifactLine`.

In a similar fashion, the `commentType` variable is set to the entry type when the first comment artifact is detected. It is then used to set the `Token`'s type, to the known comment type, of all subsequently generated comment tokens until we reach the end of the artifact line. Remember the importance of this entire process is that we identify the artifacts extracted for each line of assembly source code. The `verbose` variable is a Boolean flag that indicates if more information should be logged.

The code on lines 16 – 20 prepares a new `TokenLine` object to hold the `Tokens` that are created from the `ArtifactLine`'s payload of `Artifact` objects. Note that special care is taken to preserve the source object on line 18. You can actually get to the original assembly source code by accessing the source `Artifact`'s `source` field. In this way we preserve a connection at all times to the original line of assembly source code. Lastly, on line 23, we iterate over the line's payload. In the following sections we'll take a look at the different comparisons that are performed in more detail.

*Listing 16-6. TokenerThumb.java Main Method Details 3*

```
//Check for starting string match
023 for (Artifact art : line.payload) {
024     current = art;
025     found = false;
026     payload = art.payload;
027
028     if (inComment) {
029         compareType = commentType;
030         found = true;
031     } else {
032         for (JsonObjIsEntryType type : types.is_entry_types) {
033             boolean lfound = false;
034             int withStartsLen = 0;
035             int withEndsLen = 0;
036             int withContainsLen = 0;
037             String payloadContains = "";
038             compareType = type;
039
040             //Check for starting string match
041             for (String withStarts : type.txt_match.starts_with) {
042                 compare = withStarts;
043                 withResStarts = withStarts;
044                 withStartsLen = withStarts.length();
045                 if (withStarts.equals(JsonObjTxtMatch.special_wild_card))
046                 {
047                     //Match anything
048                     lfound = true;
049                     break;
```

```

050             } else if (withStarts.length() > 1 &&
051                 withStarts.contains(JsonObjTxtMatch.special_range)) {
052                     if (Character.isDigit(withStarts.charAt(0))) {
053                         //Found numeric range
054                         int[] range = Utils.GetIntsFromRange(withStarts);
055                         int j = 0;
056                         char lc = payload.charAt(0);
057                         try {
058                             j = Utils.GetIntFromChar(lc);
059                             if (Character.isDigit(lc) && j >= range[0] && j
060                                 <= range[1]) {
061                                 withStarts = (j + "");
062                                 withStartsLen = 1;
063                                 lfound = true;
064                                 break;
065                             }
066                         } catch (ExceptionMalformedRange e) {
067                             //do nothing
068                         }
069
070             } else if
071                 (withStarts.equals(JsonObjTxtMatch.special_lowercase_range)) {
072                     //Found lower case character range
073                     char lc = payload.charAt(0);
074                     if (Character.isLowerCase(lc)) {
075                         withStarts = (lc + "");
076                         withStartsLen = 1;
077                         lfound = true;
078                         break;
079                     }
080             } else if
081                 (withStarts.equals(JsonObjTxtMatch.special_lowercase_num_range)) {
082                     //Found lower case character range numeric
083                     char lc = payload.charAt(0);
084                     if (Character.isLowerCase(lc) ||
085                         Character.isDigit(lc)) {
086                         withStarts = (lc + "");
087                         withStartsLen = 1;
088                         lfound = true;
089                         break;
090                     }

```

```

088             } else if
(withStarts.equals(JsonObjTxtMatch.special_uppercase_range)) {
089                     //Found upper case character range
090                     char lc = payload.charAt(0);
091                     if (Character.isUpperCase(lc)) {
092                         withStarts = (lc + "");
093                         withStartsLen = 1;
094                         lfound = true;
095                         break;
096                     }
097
098             } else if
(withStarts.equals(JsonObjTxtMatch.special_uppercase_num_range)) {
099                     //Found upper case character range numeric
100                     char lc = payload.charAt(0);
101                     if (Character.isUpperCase(lc) ||
Character.isDigit(lc)) {
102                         withStarts = (lc + "");
103                         withStartsLen = 1;
104                         lfound = true;
105                         break;
106                     }
107
108             }
109         } else {
110             if (payload.indexOf(withStarts) == 0) {
111                 lfound = true;
112                 break;
113             }
114         }
115     }
116 ...

```

A listing from the TokenerThumb class' LineTokenize method, specifically the "starts with" comparisons, showing the main processing loop. The loop starts with the in-comment case followed by the entry type scan. Part 2 of the method review.

We'll pick up the method review at line 23, at the start of the main loop. On lines 24 – 26, some local variables are reset for the current loop iteration. Next on lines 28 – 31, the simple and direct case of when we're in a comment

is handled. The `compareType` variable is set to that of the `commentType` and the `found` Boolean flag is set to true. If we're not in a comment, then the block of code starting on line 32 executes.

In order to find a match for the current artifact we have to scan through all the supported entry types. On lines 33 – 38 a new set of local variables is prepared for use on the series of comparisons performed to identify the entry type. A new, local, found flag, `lfound`, is declared on line 33. The next three local variables `withStartsLen`, `withEndsLen`, and `withContainsLen` are used to track the length of the string that is used in the comparison.

The `payloadContains` variable, line 37, is used in processing the `contains` string check. The last entry in the block, line 38, sets the `compareType` variable to the type of the current loop iteration. The code on lines 41 – 155 is responsible for handling the “starts with” string comparison. The following checks are performed in this block of code. The code on lines 42 – 44 is there to reset the local variables for this loop iteration. Note that we are setting the string to be compared with, the start character, and the length of the available comparison. A summary of the comparisons made in identifying an entry type are as follows.

#### ***Listing 16-7. TokenerThumb.java Main Method Details 4***

- Match anything – special character match
- Found numeric range
- Found lower case character range
- Found lower case alpha-numeric range
- Found upper case character range
- Found upper case alpha-numeric range
- Found first character match

A list of the checks performed for the starts with, ends with, and contains comparisons.

The remaining two comparison types, must-contain and must-not-contain, are simple comparisons that do not support ranges. As such they are much more concise and less complex in their implementation. We'll take a look at the more complex comparisons first. On line 41 we iterate over the list

of different starting strings. Some of these strings will be encoded ranges, numeric or character ranges. The wildcard, “\*”, case, lines 45 – 50, is the simplest check performed. It matches on anything, thus `lfound` is set to true and the loop is exited, lines 47 – 48.

The next block of code, lines 50 – 109, is responsible for handling the more complex, range-based, comparisons. On lines 51 – 68 the numeric range is addressed. The start and stop values are compared to the current character’s value as integers. If the current character value is between the start and stop values, line 58, then we set the `withStarts` local variable to a string representation of the number and the `withStartsLen` variable to one.

Note that supported numeric ranges are single digit. The `lfound` variable is set to true and we break out of the loop on line 61 – 62. Notice that this step is wrapped in a try – catch although we don’t do anything with the exception if one occurs. Can you guess why? In this particular case if an error occurs we assume the character we’re comparing is simply not a number and that precludes it from a number range match.

This same pattern is applied in the next few comparisons on lines 68 – 108, except in these cases we’re looking for lowercase or uppercase characters and a combination of characters and numbers. The code is a bit redundant but, take a moment to look over it and make sure you understand how the comparisons work. Take a moment to look at the `JsonObjTxtMatch` class to get an idea of the categories we’re trying to match on. Similar code will be used in the ‘end’ and ‘contains’ sections of code, as we’ll soon see. The last check made, lines 110 to 113, tests if the first character of the `withStarts` string is a non-range match.

#### *Listing 16-8. TokenerThumb.java Main Method Details 5*

```
121 //Check for ending string match
122 if (lfound) {
123     lfound = false;
124     for (String withEnds : type.txt_match.ends_with) {
125         compare = withEnds;
126         withResEnds = withEnds;
127         withEndsLen = withEnds.length();
128         if (withEnds.equals(JsonObjTxtMatch.special_wild_card)) {
129             //Match anything
```

```

130         lfound = true;
131         break;
132
133     } else if (withEnds.length() > 1 &&
134         withEnds.contains(JsonObjTxtMatch.special_range)) {
135         if (Character.isDigit(withEnds.charAt(0))) {
136             //Found numeric range
137             int[] range = Utils.GetIntsFromRange(withEnds);
138             for (int z = range[0]; z <= range[1]; z++) {
139                 String zI = (z + "");
140                 int j = 0;
141                 String lc = payload.substring(payload.length() -
142                     zI.length());
143                 try {
144                     j = Utils.GetIntFromString(lc);
145                     if (j >= range[0] && j <= range[1]) {
146                         withEnds = (j + "");
147                         withEndsLen = withEnds.length();
148                         lfound = true;
149                         break;
150                     }
151                 } catch (ExceptionMalformedRange e) {
152                     //do nothing
153                 }
154             } else if
155                 (withEnds.equals(JsonObjTxtMatch.special_lowercase_range)) {
156                     //Found lower case character range
157                     char lc = payload.charAt(payload.length() - 1);
158                     if (Character.isLowerCase(lc)) {
159                         withEnds = (lc + "");
160                         withEndsLen = 1;
161                         lfound = true;
162                         break;
163                     }
164             } else if
165                 (withEnds.equals(JsonObjTxtMatch.special_lowercase_num_range)) {
166                     //Found lower case character range
167                     char lc = payload.charAt(payload.length() - 1);
168                     if (Character.isLowerCase(lc) || Character.isDigit(lc)) {
169                         withEnds = (lc + "");

```

```

169         withEndsLen = 1;
170         lfound = true;
171         break;
172     }
173
174     } else if
(withEnds.equals(JsonObjTxtMatch.special_uppercase_range)) {
175         //Found upper case character range
176         char lc = payload.charAt(payload.length() - 1);
177         if (Character.isUpperCase(lc)) {
178             withEnds = (lc + "");
179             withEndsLen = 1;
180             lfound = true;
181             break;
182         }
183
184     } else if
(withEnds.equals(JsonObjTxtMatch.special_uppercase_num_range)) {
185         //Found lower case character range
186         char lc = payload.charAt(payload.length() - 1);
187         if (Character.isUpperCase(lc) || Character.isDigit(lc)) {
188             withEnds = (lc + "");
189             withEndsLen = 1;
190             lfound = true;
191             break;
192         }
193
194     }
195 } else {
196     int compStart = 0;
197     if (payload.length() > 1) {
198         compStart = 1;
199     }
200
201     if (verbose) {
202         Logger.wrl("IndexOf withEnds: " + withEnds + " in string
payload: " + payload + " with compStart: " + compStart + " is " +
(payload.indexOf(withEnds, compStart)) + " - or - " +
(payload.lastIndexOf(withEnds)) + ", " + withStartsLen + ", compared to "
+ (payload.length() - withEndsLen) + " withStartsLen: " + withStartsLen);
203     }
204

```

```

205         if (payload.indexOf(withEnds, compStart) == (payload.length()
- withEndsLen)) {
206             lfound = true;
207             break;
208         } else if (payload.lastIndexOf(withEnds) == (payload.length()
- withEndsLen)) {
209             lfound = true;
210             break;
211         }
212     }
213 }
214 } ...

```

A listing from the TokenerThumb class' LineTokenize method showing the main processing loop, specifically the "ends with" comparisons. Part 3 of the method review.

Notice that on line 122 we don't proceed into the "ends with" comparisons unless the "starts with" comparisons were successful. In this way we don't do any unnecessary work. You'll notice that the same range comparisons are performed in this block of code, just on a slightly different string – the end character, as in the previous section we've reviewed. This code should be very similar, I'll leave it up to you to review it. Make sure you understand it before moving on. We'll see this code again in the contains comparison code block. Let's take a look at that block of code next.

#### ***Listing 16-9. TokenerThumb.java Main Method Details 6***

```

220 //Check for containing string match
221 if (lfound) {
222     lfound = false;
223     if (payload.length() >= withStartsLen + withEndsLen) {
224         payloadContains = payload.substring(withStartsLen,
(payload.length() - withEndsLen));
225     } else {
226         payloadContains = payload;
227     }
228

```

```

229     if (type.txt_match.contains == null ||
type.txt_match.contains.isEmpty() || Utils.IsEmpty(payloadContains))
{
230         lfound = true;
231     } else {
232         char[] chars = payloadContains.toCharArray();
233         boolean llfound = false;
234
235         for (char c : chars) {
236             llfound = false;
237             for (String withContains : type.txt_match.contains) {
238                 compare = withContains;
239                 withResContains = withContains;
240                 withContainsLen = withContains.length();
241
242                 if
(withContains.equals(JsonObjTxtMatch.special_wild_card)) {
243                     //Match anything
244                     llfound = true;
245                     break;
246
247                 } else if (withContains.length() > 1 &&
withContains.contains(JsonObjTxtMatch.special_range)) {
248                     if (Character.isDigit(withContains.charAt(0))) {
249                         //Found numeric range
250                         int[] range = Utils.GetIntsFromRange(withContains);
251                         for (int z = range[0]; z <= range[1]; z++) {
252                             String zI = (z + "");
253                             int j = 0;
254                             String lc =
payloadContains.substring(payloadContains.length() - zI.length());
255                             try {
256                                 j = Utils.GetIntFromString(lc);
257                                 if (j >= range[0] && j <= range[1]) {
258                                     withContains = (j + "");
259                                     withContainsLen = withContains.length();
260                                     llfound = true;
261                                     break;
262                                 }
263                             } catch (ExceptionMalformedRange e) {
264                                 //do nothing
265                             }
266                         }

```

```

267
268         } else if
269             (withContains.equals(JsonObjTxtMatch.special_lowercase_range)) {
270                 //Found lower case character range
271                 if (Character.isLowerCase(c)) {
272                     withContains = (c + "");
273                     withContainsLen = 1;
274                     llfound = true;
275                     break;
276                 }
277
278         } else if
279             (withContains.equals(JsonObjTxtMatch.special_lowercase_num_range)) {
280                 //Found lower case character range
281                 if (Character.isLowerCase(c) ||
282                     Character.isDigit(c)) {
283                     withContains = (c + "");
284                     withContainsLen = 1;
285                     llfound = true;
286                     break;
287                 }
288
289         } else if
290             (withContains.equals(JsonObjTxtMatch.special_uppercase_range)) {
291                 //Found upper case character range
292                 if (Character.isUpperCase(c)) {
293                     withContains = (c + "");
294                     withContainsLen = 1;
295                     llfound = true;
296                     break;
297                 }
298
299         } else if
300             (withContains.equals(JsonObjTxtMatch.special_uppercase_num_range)) {
301                 //Found lower case character range
302                 if (Character.isUpperCase(c) ||
303                     Character.isDigit(c)) {
304                     withContains = (c + "");
305                     withContainsLen = 1;
306                     llfound = true;
307                     break;
308                 }

```

```

304
305
306     } else {
307         if ((c + "").equals(withContains)) {
308             l1found = true;
309             break;
310         }
311     }
312 }
313 }
314
315     if (l1found) {
316         lfound = true;
317     }
318 }
319 } ...

```

A listing from the TokenerThumb class' LineTokenize method showing the main processing loop, specifically the "contains" comparisons. Part 4 of the method review.

The last big block of code begins on line 221 where we check to see if the previous comparisons resulted in a true result. If not, there's no point running the checks in this code block because we already know we don't have a match. Remember that we checked for the starts with matches, then the ends with matches, now we are checking for the contains matches. If things are a little unclear at this point take a moment to look at the `is_entry_types.json` file and you'll see the matching rules associated with each entry type.

The variable `lfound` is reset on line 222 and the string that we're working with in this section is prepared on lines 223 – 227. Notice that the starts with and ends with lengths are used to define the contains comparison string. We want to trim off the starting and ending characters that we've already matched, line 224. On lines 229 – 231 we check to see if the comparison string is empty and if so, we set `lfound` to true on line 230. This is because there's nothing left to check and our start with and end with matches were a success.

A similar set of range comparisons are performed in the code block on lines 235 – 313. I won't review this material again in any detail because it's so similar to the code we've already reviewed. Take a moment to look over it on your own and make sure you understand it before moving on.

*Listing 16-10. TokenerThumb.java Main Method Details 7*

```
325 //Check for must contain
326 if (!found && !Utils.IsEmpty(type.txt_match.must_contain)) {
327     found = true;
328     for (String s : type.txt_match.must_contain) {
329         if (!payload.contains(s)) {
330             found = false;
331             break;
332         }
333     }
334 }
335
336 if (verbose) {
337     Logger.wrl(current.payload + " Compare: '" + compare + "'"
338     MustNotContain result: " + found + " CompareType: " +
339     compareType.type_name);
340 }
341
342 //Check for must not contain
343 if (!found && !Utils.IsEmpty(type.txt_match.must_not_contain)) {
344     found = true;
345     for (String s : type.txt_match.must_not_contain) {
346         if (payload.contains(s)) {
347             found = false;
348             break;
349         }
350     }
351 if (verbose) {
352     Logger.wrl(current.payload + " Compare: '" + compare + "'"
353     MustNotContain result: " + found + " CompareType: " +
354     compareType.type_name);
355 }
356
357 if (found) {
```

```
356     found = true;  
357     break;  
358 } ...
```

A listing from the `TokenerThumb` class' `LineTokenize` method showing the last two remaining comparisons, must contain and must not contain. Part 5 of the method review.

The remaining two comparisons, shown in the previous listing, are used to test the “must contain” and “must not contain” rules. Remember that we’re using these text processing rules to find an entry type that matches the text contained in the current `Artifact`. Once we found a match, we can then create a new `Token` object to replace the `Artifact`. Let’s take a look at the JSON object that powers this aspect of the assembler.

The “contains” rules are much simpler to process than previous matching rules we’ve looked at prior. In this section the operative checks occur on lines 329 and 344. The surrounding code simply loops over the “must contain” and “must not contain” arrays and search for a match or ensures no match exists. I think the code is straight forward so we’ll omit a detailed review here.

*Listing 16-11. TokenerThumb.java Main Method Details 8*

```
{  
    "obj_name": "is_entry_type",  
    "type_name": "Comment",  
    "type_category": "Comment",  
    "category": "Comment",  
    "category_class": "java.lang.String",  
    "txt_match": {  
        "starts_with": [";"],  
        "contains": ["*"],  
        "must_contain": [],  
        "must_not_contain": [],  
        "ends_with": ["*"]  
    }  
}
```

An example is\_entry\_type JSON object demonstrating the attributes used in the text comparisons we've just reviewed.

Take a look at the JSON object in the previous listing. Note that the attributes of the text match object are used to find a matching entry type. We've been reviewing the code that powers each of the comparisons in the text match object. At the conclusion of the series of comparisons if we've found a match then we can create a new `Token` object instance to represent the `Artifact` that we're processing. The code in the next listing shows the creation of a new `Token` object.

*Listing 16-12. TokenerThumb.java Main Method Details 9*

```
362 Token tmb = new Token();
363 tmb.artifact = art;
364 tmb.index = count;
365 tmb.lineNumAbs = lineNum;
366 tmb.source = payload;
367 if (!found || compareType == null) {
368     tmb.type_name = "Unknown";
369     tmb.type = null;
370 } else {
371     tmb.type_name = compareType.type_name;
372     tmb.type = compareType;
373 }
374 tmb.payload = new ArrayList<>();
375
376 ret.payload.add(tmb);
377 count++;
378
379 if (tmb.type_name.equals(JsonObjIsEntryTypes.NAME_OPCODE)) {
380     tmb.isOpCode = true;
381
382 } else if (tmb.type_name.equals(JsonObjIsEntryTypes.NAME_DIRECTIVE)) {
383     tmb.isDirective = true;
384
385 } else if (tmb.type_name.equals(JsonObjIsEntryTypes.NAME_LABEL)) {
386     tmb.isLabel = true;
387 }
```

```
388 } else if (tmb.type_name.equals(JsonObjIsEntryTypes.NAME_LABEL_REF)) {  
389     tmb.isLabelRef = true;  
390  
391 } else if (tmb.type_name.equals(JsonObjIsEntryTypes.NAME_COMMENT)) {  
392     tmb.isComment = true;  
393     commentType = compareType;  
394     inComment = true;  
395  
396 } else if  
(tmb.type_name.equals(JsonObjIsEntryTypes.NAME_DIRECTIVE_STRING)) {  
397     tmb.source = tmb.source.replace("|", "");  
398  
399 }
```

A listing from the TokenerThumb class' LineTokenize method showing the creation of a Token based on the current identified Artifact being processed. Part 6 of the method review.

The last block of code for us to look at, shown in the previous listing, is responsible for converting the `Artifact` being processed into a `Token`. On lines 362 – 366 the token is created, and some fields are set. The original source code and artifact are set on lines 363 and 366 respectively. On lines 367 – 373 we set the `type` and `type_name` fields of the `Token` class. If we haven't been able to find a matching entry type the resulting type name will be "Unknown." This should not happen and is an indication one of the text match rules may be incorrect for a given entry type.

On line 374 the `payload` for the token is prepped. Remember that in certain situations sub-tokens, tokens belonging to a `Token` object and not a `TokenLine` object, are used. The new `Token` is added to the `TokenLine`'s `payload` and the token count is incremented on lines 376 – 377. The remaining code, lines 379 – 399, is used to set certain Boolean flags that indicate a when we've identified important token types.

That brings us to the conclusion of this review section. This is an important method. Make sure you understand what's going on here. Can you think of a simpler, more powerful way to implement this feature of the assembler? For instance, instead of using the text match JSON object as

we've done so here, what could we do that would simplify this part of the assembly process? One idea might be to support regex. We could add a regex string to the text match object and rewrite this method to run one simple regex rule and process the result. Food for thought. In the next section we'll see the class in action.

## Demonstration: TokenerThumb.java

To demonstrate this class in action we're going to do something a little different than with previous class reviews. The code example associated with this class would just be a simple call to the `FileTokenize` method and wouldn't be that interesting.

Instead, I want us to look at the transformation of data inside the assembler as it processes a line of source code. In the next three listings you'll see an example line of source code as it goes from text to an `ArtifactLine` object and finally a `TokenLine` object. With each permutation we add new information to the objects that the assembler uses as you can see in the length of the `TokenLine` JSON object listed subsequently. Let's take a look.

*Listing 16-13. TokenerThumb.java Demonstration 1*

```
1      test          @EQU      2
```

A sample line of Thumb-1 assembly source code.

The first example, listed previously, shows the original assembly source code.

*Listing 16-14. TokenerThumb.java Demonstration 2*

```
{
  "obj_name": "ArtifactLine",
  "lineNum": 5,
  "sourceLen": 25,
  "source": "      test          @EQU      2",
```

```

"payload": [
  {
    "obj_name": "Artifact",
    "posStart": 4,
    "posStop": 7,
    "len": 4,
    "lineNum": 5,
    "index": 0,
    "payload": "test"
  },
  {
    "obj_name": "Artifact",
    "posStart": 16,
    "posStop": 19,
    "len": 4,
    "lineNum": 5,
    "index": 1,
    "payload": "@EQU"
  },
  {
    "obj_name": "Artifact",
    "posStart": 24,
    "posStop": 24,
    "len": 1,
    "lineNum": 5,
    "index": 2,
    "payload": "2"
  }
]
}

```

An example of the previously shown line of assembly source code converted to an `ArtifactLine` object.

The next example shows the `TokenLine` object and the extracted `Tokens`. Notice that the original source code is available at this level. We can view the `ArtifactLine`, the assembly source code, and each `Artifact` as well as the `Tokens` we've identified from the given `Artifact`. The full trace, so to speak, from assembly source code to Token is available in the hierarchy of objects. Take a moment to look it over. Some entries have been simplified, abridged, for brevity.

*Listing 16-15. TokenerThumb.java Demonstration 3*

```
{  
    "obj_name": "TokenLine",  
    "byteLength": 2,  
    "addressInt": 0,  
    "lineNumActive": 0,  
    "lineNumAbs": 5,  
    "payloadLen": 3,  
    "payloadLenArg": 0,  
    "isLineEmpty": false,  
    "isLineOpCode": false,  
    "isLineDirective": false,  
    "isLineLabelDef": false,  
    "source": {  
        "obj_name": "ArtifactLine",  
        "lineNum": 5,  
        "sourceLen": 25,  
        "source": "    test      @EQU    2",  
        "payload": [  
            {  
                "obj_name": "Artifact",  
                "posStart": 4,  
                "posStop": 7,  
                "len": 4,  
                "lineNum": 5,  
                "index": 0,  
                "payload": "test"  
            },  
            {  
                "obj_name": "Artifact",  
                "posStart": 16,  
                "posStop": 19,  
                "len": 4,  
                "lineNum": 5,  
                "index": 1,  
                "payload": "@EQU"  
            },  
            {  
                "obj_name": "Artifact",  
                "posStart": 24,  
                "posStop": 24,  
                "len": 1,  
                "lineNum": 5,  
                "index": 2,  
                "payload": ""  
            }  
        ]  
    }  
}
```

```

    "lineNum": 5,
    "index": 2,
    "payload": "2"
  }
]
},
"payload": [
{
  "obj_name": "Token",
  "type_name": "Label",
  "source": "test",
  "lineNumAbs": 5,
  "index": 0,
  "payloadLen": 0,
  "payloadArgLen": 0,
  "isOpCodeArg": false,
  "isDirectiveArg": false,
  "isOpCode": false,
  ...
  "isLabelRef": false,
  "artifact": {
    ...
  },
  "type": {
    ...
    ,
    "name": "net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsEntryType",
    "fileName": "./cfg/THUMB/is_entry_types.json",
    "loader": "net.middlemind.GenAsm.Loaders.Thumb.LoaderIsEntryTypes"
  },
  "payload": []
},
{
  "obj_name": "Token",
  "type_name": "Directive",
  "source": "@EQU",
  "lineNumAbs": 5,
  "index": 1,
  "payloadLen": 0,
  "payloadArgLen": 0,
  "isOpCodeArg": false,
  ...
  "isLabelRef": false,

```

```

"artifact": {
    ...
},
"type": {
    ...
,
    "name": "net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsEntryType",
    "fileName": "./cfg/THUMB/is_entry_types.json",
    "loader": "net.middlemind.GenAsm.Loaders.Thumb.LoaderIsEntryTypes"
},
"payload": []
},
{
    "obj_name": "Token",
    "type_name": "Number",
    "source": "2",
    "lineNumAbs": 5,
    "index": 2,
    "payloadLen": 0,
    "payloadArgLen": 0,
    "isOpCodeArg": false,
    ...
    "isLabelRef": false,
    "artifact": {
        ...
    },
    "type": {
        ...
        ,
        "name": "net.middlemind.GenAsm.JsonObjs.Thumb.JsonObjIsEntryType",
        "fileName": "./cfg/THUMB/is_entry_types.json",
        "loader": "net.middlemind.GenAsm.Loaders.Thumb.LoaderIsEntryTypes"
    },
    "payload": []
}
]
}

```

An abridged listing of a TokenLine object in JSON form.

The previous example shows a summary of a `TokenLine` object. As you can see you can access the `ArtifactLine` it's based on, and even access the original line of source code if need be. Notice that each `Token` now has a type associated with it. This is a good example of how you would debug a new instruction set. Parsing each instruction, one at a time, and making sure the tokenizer is identifying artifacts correctly. That brings us to the conclusion of this class review. Let's summarize what we've covered in this chapter.

## Chapter Conclusion

In this chapter we took a look at the Thumb-1 implementation of the `Tokener` interface. The implementation class, `TokenerThumb`, is used to identify the `Artifacts` extracted from a line of assembly source code and associate them with a known entry type in a `Token` object. Let's summarize some of the key topics we've reviewed in this chapter.

- **TokenerThumb Review:** We did an in-depth review of the `TokenerThumb` class covering the different comparisons performed to identify `Artifacts`.
- **Assembly Source to TokenLine JSON Object:** We also got to see the progression a line of source code takes as it goes through the lexer and subsequently the tokenizer. This demonstration should provide a solid idea of the work that's going on underneath the hood when these classes are in use.

We've come a long way from where we started, reviewing basic assembly source code and opcode documentation. In the next chapter we'll look at the most challenging part of the project, the assembler. Basically, what the assembler does is convert a line of assembly source code into a `TokenLine` object then attempt to find a matching opcode from the instruction set's data files. Of course, it's all a bit more complicated than that. We have our work cut out for us. Let's get to it, shall we?

# Chapter 17: The Assembler Part 1

Take a moment to pat yourself on the back. You've made your way through a ton of material to get to this point. In Chapter 17 we begin the review of the assembler. This is no small task. We have a lot of code to look at. As you might have surmised, we'll be reviewing the `AssemblerThumb` class, the ARM Thumb-1 instruction set implementation of the `Assembler` interface.

I want to list the major steps the `AssemblerThumb` class takes to complete the assembly process here. Take a moment to review it and make sure you have an idea of what each step does. We'll review the class methods in detail, so you'll understand exactly how the assembler functions and accomplishes each step. Please note there are a fair amount of steps. This is by design.

The GenAsm assembler is designed more for teaching purposes than anything else. As such, we perform some operations in independent steps rather than one larger complex step. We also write a ton of output files. These are to help with understanding the flow of data from the assembly source code file to the final binary representation.

- **STEP 1:** Process JsonObjIsSet's file entries and load then parse the JSON object data.  
Operative Method: LoadAndParseJsonObjData
- **STEP 2:** Link loaded JSON object data to any already loaded sub-objects, Java instances.  
Operative Method: LinkJsonObjData
- **STEP 3:** Load and lexerize the assembly source file creating `ArtifactLines` that contain `Artifacts` representing the assembly source code.  
Operative Method: LexerizeAssemblySource
- **STEP 4:** Tokenize the lexerized artifacts. By processing text matching rules we create `TokenLines` that contain `Tokens` from the `ArtifactLines` and `Artifacts` that were created during the lexerization process.  
Operative Method: TokenizeLexerArtifacts
- **STEP 5:** Validate `TokenLines` against loaded JSON data that describes what are valid lines for the given instruction set implemention.  
Operative Method: ValidateTokenizedLines
- **STEP 6:** Combine comment `Tokens` as children of the initial comment `Token`.  
Operative Method: CollapseCommentTokens
- **STEP 7:** Expand register ranges into individual register entries.  
Operative Method: ExpandRegisterRangeTokens
- **STEP 8:** Combine list and group `Tokens` as children of the initial list or group `Token`.  
Operative Method: CollapseListAndGroupTokens
- **STEP 9:** Mark OpCode, OpCode argument, and register `Tokens`.  
Operative Method: PopulateOpCodeAndArgData

- **STEP 10:** Mark directive and directive argument `Tokens`, create area-based line lists with hex numbering.  
Operative Method: `PopulateDirectiveArgAndAreaData`
- **STEP 11:** Validate OpCode lines against known OpCodes by comparing arguments.  
Operative Method: `ValidateOpCodeLines`
- **STEP 12:** Validate directive lines against known directives by comparing arguments.  
Operative Method: `ValidateDirectiveLines`
- **STEP 13:** List assembly source areas and build binary output.  
Write Area Source Code Lines: `output_area_lines_code.json`  
Write Area Source Code Desc: `output_area_desc_code.json`  
Operative Method: `BuildBinLines`

We'll follow this exact series of steps during the class' main method review. In the next section we'll start the review with an overview of the class and the review strategy. Let's take a look.

## Class Review: AssemblerThumb.java

The `AssemblerThumb` class is responsible for running the lexer, tokenizer, and assembling the source code into a binary representation of the code and data areas, if defined, in the source code files. The sections we'll use to review the class are listed here:

- Static Class Members
- Class Fields
- Pertinent Method Outline/Class Headers
- Main Method Details
- Demonstration

To start off the review we'll look at the class' static members followed by a look at the class fields. This is a long and complex class we have to

review. Don't expect to absorb everything in one reading. It's perhaps better to return to this chapter when you need to refresh your memory.

## Static Class Members: AssemblerThumb.java

The AssemblerThumb class has a few static class fields for us to review, shown in the following listing.

*Listing 17-1. AssemblerThumb.java Static Class Fields*

```
public static String ENDIAN_NAME_BIG = "BIG";
public static String ENDIAN_NAME_LITTLE = "LITTLE";
public static String ENDIAN_NAME_JAVA_DEFAULT = "BIG";
public static String INSTRUCTION_ALIGNMENT_NAME_WORD = "HALFWORD";

public static int INSTRUCTION_ALIGNMENT_BYTES = 2;
public static int INSTRUCTION_ALIGNMENT_BITS = 16;
public static String NUMBER_SHIFT_NAME_LEFT = "LEFT";
public static String NUMBER_SHIFT_NAME_RIGHT = "RIGHT";
public static String SPECIAL_ADD_OP_CODE_CHECK = "101100000";
```

The AssemblerThumb class' static class fields.

The class' static fields provide a lookup for some high-level information regarding big- and little- endian encoding, shift types, and alignment. The first entry, `ENDIAN_NAME_BIG`, is a string representation of the endian encoding followed by a similar representation of the little-endian encoding. The next field, `ENDIAN_NAME_JAVA_DEFAULT`, holds a string representation of Java's default endian encoding. This comes in handy in certain situations like when handling the one's complement etc.

The next field, similarly, provides a string representation of the instruction set's alignment followed by the number of bytes and bits in an instruction set. The next two fields, `NUMBER_SHIFT_NAME_LEFT` and `NUMBER_SHIFT_NAME_RIGHT` provide a string representation of the different shift types. The last entry, `SPECIAL_ADD_OP_CODE_CHECK`, is a piece of a binary string representing a special ADD opcode check.

It is a bit of an anomaly so I'll talk about it before we move on. We have to handle a specific case of the ADD opcode that requires us to toggle one bit of the binary representation but only after we know the value expressed as an argument is positive or negative. It's easier to store the new opcode binary string than to calculate the adjustment so that's what we did. We'll see this come up in the class method review. In the next section we'll finish reviewing the remaining class fields.

## Class Fields: AssemblerThumb.java

The AssemblerThumb class has a number of class fields that we have to review. This process will give us a good idea of how the fields are used so that we can further refine our understanding of them during the class' method review sections and the demonstration section. There are a fair number of class fields for us to review so I'll break things out into three groups. I'll list the first group of class fields here.

*Listing 17-2. AssemblerThumb.java Class Fields 1*

```
public String obj_name = "AssemblerThumb";
public JsonObjIsSet isaDataSet;
public Map<String, JsonObj> isaData;
public Map<String, Loader> isaLoader;
public Map<String, String> jsonSource;
public JsonObjIsEntryTypes jsonObjIsEntryTypes;
public JsonObjIsValidLines jsonObjIsValidLines;
public JsonObjIsEmptyDataLines jsonObjIsEmptyDataLines;
public JsonObjIsOpCodes jsonObjIsOpCodes;
public JsonObjIsDirectives jsonObjIsDirectives;
public JsonObjRegisters jsonObjRegisters;
public String asmSourceFile;
public List<String> asmDataSource;
public List<ArtifactLine> asmDataLexed;
```

A listing of the AssemblerThumb class' fields, part 1.

The first set of fields for us to review starts off with the all too familiar `obj_name` field. The proper name for this class, following the convention, is 'AssemblerThumb'. The next field, `isaDataSet`, is an instance of the

`JsonObjIsSet` class and is meant to hold the instruction set definition used by this assembler implementation. The next three fields aren't used much in the assembler code, but they are here for completeness and bookkeeping.

The `isaData`, `isaLoader`, and `jsonSource` fields are all `Map` instances that are used to store the holder, loader, and JSON source of the different data files used by the assembler. This way we can quickly look them up if need be. The `isaData` field stores loaded JSON data files stored by object name, using the `obj_name` field. The `isaLoader` field holds loader class instances stored by the full class name, using the `loader_class` field. Lastly the `jsonSource` field stores the assembly source code files by full path name, `path`. The next six fields are holder class instances and are used to keep a reference to the holder class for the following JSON data files, entry types, valid lines, empty data lines, opcodes, directives, and registers. Take a moment to look of the JSON data files to refresh your memory and see what they contain.

This is a bit of an overkill since we have the same data stored in the `isaData` field, but this approach has the added benefit of being slightly more efficient, so we allow it. The next field `entry`, `asmSourceFile`, holds the full path to the specified assembly source code file. The next field, `asmDataSource`, holds the results of loading the assembly source code file, a list of strings. If you think back to our review of the assembly process, you should recognize the next field, `asmDataLexed`, as being the output from the lexer.

The assembly source code has been converted into `ArtifactLines` with a payload, `list`, of `Artifact` objects representing the key aspects of each line of assembly source code. Can you think of a field we might see next, that's not shown here? If you thought about a list that holds the output of the tokenizer, a list of `TokenLine` objects, then you would be right. Let's look at the next group of class fields now.

### *Listing 17-3. AssemblerThumb.java Class Fields 2*

```
public List<TokenLine> asmDataTokened;
public Symbols symbols;
public List<String> requiredDirectives;
public AreaThumb areaThumbCode;
```

```
public AreaThumb areaThumbData;
public List<TokenLine> asmAreaLinesCode;
public List<TokenLine> asmAreaLinesData;
public int lineLenBytes = 2;
public int lineLenHalfWords = 1;
public int lineLenWords = 0;
public JsonObjBitSeries lineBitSeries;
public JsonObjNumRange lineNumRange;
public int pcPreFetchBytes;
public int pcPreFetchWords;
```

A listing of the AssemblerThumb class' fields, part 2.

The first entry, as you may have guessed, is a list of `TokenLine` objects, the `asmDataTokened` field. Following this entry is the `symbols` field used to track and maintain the symbols used throughout the assembly source code. The next few fields start getting into the structure of an assembly program as supported by this assembler implementation.

In short, our assembler only supports two area types. An area is just that, a designated area of an assembly source code file. The two types of areas supported are the `@DATA` and `@CODE` areas. A second limitation, done so to simplify some of the process, is that the default linker, as implemented for this instruction set, can only link a maximum of two areas, and those areas have to be defined in the same file.

This is fine. We are still able to write some fairly robust example programs and the simplicity serves to keep the code attainable and the code review just that much shorter. The `requiredDirectives` class field is used to track the directives that are required to define the proper structure of an assembly program. Because we know there's a maximum of two areas that can exist, we have two class fields, `areaThumbCode` and `areaThumbData`, that track the code and data areas respectively.

The subsequent fields, `asmAreaLinesCode` and `asmAreaLinesData`, are lists that hold references to the actual `TokenLines` associated with each area, code and data. The following three fields, `lineLenBytes`, `lineLenHalfwords`, `lineLenWords`, are used to

provide an integer representation of the implemented instruction set's line length. The `lineBitSeries` and `lineNumRange` fields are used to hold information about the instruction set's line length as defined by the JSON data files.

Currently we hold information about the length of an instruction's binary representation in a few places. We have some static class fields that describe a line, we have some class fields with default values, and we also have some data loaded in from the JSON data files that describe the binary length of a line of assembly code in our instruction set implementation.

The last two class fields listed in this group are the `pcPreFetchBytes` and `pcPreFetchWords` fields. These fields are similar to the line length fields we just reviewed but are used to track the length of the prefetch in bytes and words. Although not really referenced in our implementation we load up the data that describes our instruction set, Thumb-1, in as complete a way as possible. Prefetch describes how the hardware, or in our case the emulated hardware, handles looking up the next instruction in the program. We sometimes have to take this into consideration when calculating how to reach assembly data or code that is referenced by a symbol. The last group of class fields for us to look at is as follows.

*Listing 17-4. AssemblerThumb.java Class Fields 3*

```
public int pcPreFetchHalfwords;
public Integer asmStartLineNumber;
public boolean isEndianBig = false;
public boolean isEndianLittle = true;
public TokenLine lastLine;
public Token lastToken;
public int lastStep;
public String assemblyTitle;
public String assemblySubTitle;
public Object other;
public String rootOutputDir;
public AssemblerEventHandlerThumb eventHandler;
public boolean verboseLogs = false;
public boolean quellFileOutput = false;
```

A listing of the `AssemblerThumb` class' fields, part 3.

A third measure of the prefetch length, `pcPrefetchHalfwords`, is used to provide another convenient measure, this time in halfwords, or shorts. The `asmStartLineNumber` field is used in conjunction with the `@ORG` directive and is used to set the starting line number, memory location, of the assembly program as it would be run on the target hardware. We'll see this in action when we review the sample programs.

The subsequent two fields, `isBigEndian` and `isLittleEndian`, are used to denote the default endianness of the assembler as indicated by the JSON data files. Even though we generate both types of binary output by default we want to establish a baseline for the assembler. The next three fields are very important. The `lastLine`, `lastToken`, and `lastStep` fields are used throughout the class to track the current work the assembler is doing in the event an exception is encountered.

The fields `assemblyTitle` and `assemblySubTitle`, are used to store the actual values associated with the assembly source code file as defined by the `@TTL` and `@SUBT` directives. The object, `other`, is used to hold process customization data, if any. We currently don't use this feature, but we keep a reference to the passed in object, just in case. The last four class fields drive the behavior of the class. The `rootOutputDir` field is used to define the path to the root of the output directory. The `eventHandler` field is used to hold a reference to an event handler class that's designed to customize certain assembly steps by handing callback events that occur before and after major steps in the assembly process.

Lastly, the `verboseLogs` and `quellFileOutput` fields are Boolean flags that control the amount of output and the generation of output files respectively. That brings us to the conclusion of the class field review. In the next section we'll take a look at the class' pertinent methods and declaration.

## Pertinent Method Outline/Class Headers: **AssemblerThumb.java**

In this section we'll take a look at the `AssemblerThumb` class' pertinent methods and class declaration including import statements and implemented interfaces. We'll take a look at the class' pertinent method outline first.

*Listing 17-5. AssemblerThumb.java Pertinent Method Outline/Class Headers 1*

```
//Support Methods
public JsonObjIsDirective FindDirectiveArgMatches(List<JsonObjIsDirective>
directiveMatches, List<Token> args, Token directiveToken);

public List<JsonObjIsDirective> FindDirectiveMatches(String directiveName,
int argLen);

public Token FindFirstDirective(TokenLine line);
public TokenLine FindNextOpCodeLine(int lineNumber, String label);
public JsonObjIsOpCode FindOpCodeArgMatches(List<JsonObjIsOpCode>
opCodeMatches, List<Token> args, Token opCodeToken);

public List<JsonObjIsOpCode> FindOpCodeMatches(String opCodeName, int
argLen);

public int CountArgTokens(List<Token> payload, int argCount);
public int CountArgTokens(List<Token> payload, int argCount, String
argCategory, boolean isOpCodeArg);

public JsonObjIsEntryType FindEntryType(String entryName);
public String CleanRegisterRangeString(String range, String rangeDelim);

public void CleanTokenSource(Token token);
public void LexerizeAssemblySource(int step);
public void TokenizeLexerArtifacts(int step);
public int[]
FindValidLineEntryWithMatchingTokenEntryType(JsonObjIsValidLine validLine,
Token token, int entry, int index);

public Object[] ProcessShift(Integer tIntIn, JsonObjNumRange num_range,
JsonObjBitShift bit_shift, JsonObjBitSeries bit_series, String source, int
lineNumAbs, boolean bitLen2x, boolean checkRange);

public Object[] ProcessSymbolValue(String source, JsonObjBitSeries
bit_series, int lineNumAbs, TokenLine line);

public void BuildBinOpCodePrep(int step, TokenLine line);

//Main Methods
```

```

public void RunAssembler(JsonObjIsSet jsonIsSet, String
assemblySourceFile, List<String> assemblySource, String outputDir, Object
otherObj, AssemblerEventHandler asmEventHandler, boolean verbose, boolean
quellOutput);

public void LoadAndParseJsonObjData(int step);
public void LinkJsonObjData(int step);
public boolean ValidateTokenizedLines(int step);
public boolean ValidateTokenizedLine(int step, TokenLine line,
JsonObjIsValidLines validLines, JsonObjIsValidLine validLineEmpty);

public void CollapseCommentTokens(int step);
public void ExpandRegisterRangeTokens(int step);
public void CollapseListAndGroupTokens(int step);
public void PopulateOpCodeAndArgData(int step);
public void PopulateDirectiveArgAndAreaData(int step);

public void ValidateOpCodeLines(int step);
public void ValidateDirectiveLines(int step);
public void CleanAreas();
public void CleanSymbols();
public void BuildBinLines(int step, List<TokenLine> areaLines, AreaThumb
area);

public void BuildBinDirective(int step, TokenLine line);
public void BuildBinOpCode(int step, TokenLine line);

```

A listing of the `AssemblerThumb` class' pertinent main and support methods.

There are a lot of methods for us to review in the `AssemblerThumb` class. No worries we'll go over each one in turn. In this next listing we'll look at the class' header. Take a moment to read over the class' declaration. Keep an eye out for any interesting import statements and any interfaces implemented by the class. The import statements for certain groups have been abridged using the “`.*`” notation.

#### ***Listing 17-6. AssemblerThumb.java Pertinent Method Outline/Class Headers 2***

```

import net.middlemind.GenAsm.Assemblers.Symbol;
import net.middlemind.GenAsm.Assemblers.Symbols;

```

```

import net.middlemind.GenAsm.AssemblersAssembler;
import net.middlemind.GenAsm.Tokeners.Thumb.TokenerThumb;
import net.middlemind.GenAsm.Tokeners.TokenLine;
import net.middlemind.GenAsm.Tokeners.Token;
import net.middlemind.GenAsm.Tokeners.TokenSorter;
import net.middlemind.GenAsm.Lexers.ArtifactLine;
import net.middlemind.GenAsm.Lexers.Thumb.LexerThumb;
import net.middlemind.GenAsm.Exceptions.Thumb.*;
import net.middlemind.GenAsm.JsonObjs.Thumb.*;
import net.middlemind.GenAsm.JsonObjs.*;
import net.middlemind.GenAsm.Loaders.Loader;
import java.io.IOException;
import java.lang.reflect.InvocationTargetException;
import java.nio.ByteBuffer;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Hashtable;
import java.util.List;
import java.util.Map;
import net.middlemind.GenAsm.AssemblersAssembler.EventHandler;

import
net.middlemind.GenAsm.Assemblers.Thumb.BuildOpCodeEntryThumbSorter.BuildOp
CodeEntryThumbSorterType;

import net.middlemind.GenAsm.FileIO.FileLoader;
import net.middlemind.GenAsm.Logger;
import net.middlemind.GenAsm.Tokeners.TokenSorterTokenSorterType;

import net.middlemind.GenAsm.Utils;

public class AssemblerThumb implements Assembler { }

```

A listing of the AssemblerThumb class' declaration. Some import groups have been abridged with the “.\*” notation.

As you can see in the previous listing the AssemblerThumb class implements the Assembler interface and uses a number of imports to get the job done. In the next section we'll start things off with a review of the class' support methods. This should start to give us an idea of how the class

functions. Let your imagination work as you read through them, and we'll see them in action in the demonstration sections and when we review the class' main methods.

## Support Method Details: AssemblerThumb.java

There are a number of support methods for us to review. I'll break up the listings a bit, so they aren't so large. The first methods up for review are the `FindDirectiveArgMatches`, `FindDirectiveMatches`, and `FindFirstDirective` methods listed as follows. Let's jump into some code!

*Listing 17-7. AssemblerThumb.java Support Method Details 1*

```
01 public JsonObjectIsDirective
02     FindDirectiveArgMatches(List<JsonObjectIsDirective> directiveMatches,
03     List<Token> args, Token directiveToken) throws ExceptionNoDirectiveFound {
04     JsonObjectIsDirectiveArg directiveArg = null;
05     Token argToken = null;
06     boolean argFound = true;
07     int directiveArgIdx = -1;
08
09     for (JsonObjectIsDirective directive : directiveMatches) {
10         directiveArg = null;
11         argToken = null;
12         argFound = true;
13
14         for (int i = 0; i < directive.args.size(); i++) {
15             directiveArg = directive.args.get(i);
16             directiveArgIdx = i;
17
18             if (i < args.size()) {
19                 argToken = args.get(i);
20             } else {
21                 argFound = false;
22                 break;
23             }
24
25             if (directiveArg != null && argToken != null) {
26                 if
(directiveArg.is_entry_types.contains(argToken.type_name)) {
```

```

25             if (!Utils.IsEmpty(directiveArg.is_arg_value)) {
26                 if (
directiveArg.is_arg_value.contains(argToken.source)) {
27                     argFound = false;
28                     break;
29                 }
30             }
31         } else {
32             argFound = false;
33             break;
34         }
35     } else {
36         argFound = false;
37         break;
38     }
39 }
40
41     if (argFound) {
42         return directive;
43     }
44 }
45
46     throw new ExceptionNoDirectiveFound("Could not find a directive that
has matching arguments for line number " + directiveToken.lineNumAbs +
"with directive '" + directiveToken.source + "' at directive argument index
" + directiveArgIdx);
47 }

01 public List<JsonObjIsDirective> FindDirectiveMatches(String
directiveName, int argLen) {
02     List<JsonObjIsDirective> ret = new ArrayList<>();
03     for (JsonObjIsDirective directive :
jsonObjIsDirectives.is_directives) {
04         if (directive.directive_name.equals(directiveName) &&
directive.arg_len == argLen) {
05             ret.add(directive);
06         }
07     }
08     return ret;
09 }

01 public Token FindFirstDirective(TokenLine line) {
02     for (Token token : line.payload) {

```

```
03     if (token.type_name.equals(JsonObjIsEntryTypes.NAME_DIRECTIVE)) {  
04         return token;  
05     }  
06 }  
07 return null;  
08 }
```

The first group of support methods to review from the AssemblerThumb class.

The first method we'll look at is the `FindDirectiveArgMatches` method. This method takes a list of directives to search through, `directiveMatches`, a list of arguments to find a match for, `args`, and the original token the directive arguments were found in, `directiveToken`. It is used to identify the correct directive entry that a given token is referring to. On lines 2 – 5, some local variables are defined. The `directiveArg` variable is a temporary holder of the current directive argument we're looking at. The `argToken` variable holds the current token argument while the `argFound`, variable is used to indicate if a matching argument was found or not. Lastly, the `directiveArgIdx` variable tracks the index of the current directive argument.

On lines 7 – 10 we iterate over the list of directives to search through and reset some loop variables. Next, on line 12 we iterate over the current directive's arguments. Lines 13 and 14 are used to reset the `directiveArg`, and `directiveArgIdx` variables based on the current loop iteration. Next on lines 16 – 21 we set the `argToken` variable, based on the arguments we're trying to match, if it's defined. If not we set the `argFound` variable to false and break out of the inner for loop. Setting the `argFound` variable to true will cause the outer loop to exit on lines 41 – 43.

The current directive's argument is compared to the provided argument on lines 23 – 38, with the method returning true if the provided argument list matches the known arguments of the current directive. If not, the method returns false. That's really all there is to the `FindDirectiveArgMatches` method. Again, this method is used to find the matching directives, `JsonObjIsDirective`, for a given token. The token has been identified as being of the directive type and this method helps us find the Java class instance that matches the specified directive.

The second method for us to review from this group is the `FindDirectiveMatches` method. This method is a shallower matching method when compared to the method we've just reviewed. The method takes two arguments, a `directiveName` string, and an integer, `argLen`, that holds the number of arguments the directive takes. The purpose of this support method is to quickly find a directive that has the same, signature, name and number of arguments. The return variable, `ret`, is initialized on line 2 of the method. As you can see this method can find more than one matching directive. Lines 3 – 7 loops over the instruction set's supported directives looking for a match on the directive name and number of arguments.

If a match is found it's added to the return variable on line 5 with the list of matching directives returned on line 8 of the method. Notice how the quick comparison method, `FindDirectiveMatches`, can be used to find all potential matches for a directive `Token` while the `FindDirectiveArgMatches` method can be used to narrow the search down to one specific directive. The last method in this group, `FindFirstDirective`, is used to scan the payload of a `TokenLine` object and locate, if any, the first directive in the line. This method comes in handy when we want to detect if a line of assembly source code starts with a directive and has to be handled as such. Let's move on and check the next set of support methods for us to review.

#### *[Listing 17-8. AssemblerThumb.java Support Method Details 2](#)*

```
001 public TokenLine FindNextOpCodeLine(int lineNumber, String label) throws
ExceptionNoOpCodeLineFound {
002     if (lineNumber + 1 < asmDataTokened.size()) {
003         for (int i = lineNumber + 1; i < asmDataTokened.size(); i++) {
004             TokenLine line = asmDataTokened.get(i);
005             if ((line.isLineOpCode || line.isLineDirective) && !
line.isLineLabelDef && !line.isEmpty) {
006                 return line;
007             }
008         }
009     }
010     throw new ExceptionNoOpCodeLineFound("Could not find an OpCode line
for label '" + label + "' on empty line " + lineNumber);
```

```

011 }

001 public JsonObjIsOpCode FindOpCodeArgMatches(List<JsonObjIsOpCode>
opCodeMatches, List<Token> args, Token opCodeToken) throws
ExceptionNoOpCodeFound {
002     JsonObjIsOpCodeArg opCodeArg = null;
003     JsonObjIsOpCodeArg opCodeArgSub = null;
004     Token argToken = null;
005     Token argTokenSub = null;
006     boolean argFound = true;
007     boolean argFoundSub = false;
008     boolean hasArgsSub = false;
009     int opCodeArgIdx = -1;
010     int opCodeArgIdxSub = -1;
011
012     for (JsonObjIsOpCode opCode : opCodeMatches) {
013         opCodeArg = null;
014         argToken = null;
015         argFound = true;
016         argFoundSub = false;
017         hasArgsSub = false;
018
019         //Sort Json OpCode arguments so that they are arg_index
ascending
020         Collections.sort(opCode.args, new JsonObjIsOpCodeArgSorter());
021
022         for (int i = 0; i < opCode.args.size(); i++) {
023             opCodeArg = opCode.args.get(i);
024             opCodeArgIdx = i;
025
026             if (i < args.size()) {
027                 argToken = args.get(i);
028             } else {
029                 argFound = false;
030                 break;
031             }
032
033             if (opCodeArg != null && argToken != null) {
034                 if (!
(opCodeArg.is_entry_types.contains(argToken.type_name))) {
035                     argFound = false;
036                     break;
037                 }

```

```

038
039         if (opCodeArg.sub_args != null &&
040             opCodeArg.sub_args.size() > 0) {
041             opCodeArgSub = null;
042             argTokenSub = null;
043             argFoundSub = true;
044             hasArgsSub = true;
045             int regRangeOffset = 0;
046
047             for (int j = 0; j < opCodeArg.sub_args.size(); j++) {
048                 opCodeArgSub = opCodeArg.sub_args.get(j);
049                 opCodeArgIdxSub = j;
050
051                 if (argToken.payload != null && (j + regRangeOffset)
052                     < argToken.payload.size()) {
053                     argTokenSub = argToken.payload.get(j +
054                     regRangeOffset);
055                 } else {
056                     argFound = false;
057                     argFoundSub = false;
058                     break;
059                 }
060
061                 if (opCodeArgSub != null && argTokenSub != null) {
062                     if
063                         (opCodeArgSub.is_entry_types.contains("RegisterRangeLow") &&
064                         argTokenSub.type_name.equals("RegisterLow")) {
065                             for (int k = j + 1; k <
066                                 argToken.payload.size(); k++) {
067                                 if
068                                     (argToken.payload.get(k).type_name.equals("RegisterLow")) {
069                                         regRangeOffset++;
070                                     }
071                                 }
072                             }
073                         }
074                     }
075                 }
076             }
077             if (argTokenSub != null && argFoundSub) {
078                 if
079                     (opCodeArgSub.is_entry_types.contains("RegisterRangeHi") &&
080                     argTokenSub.type.type_category.equals("RegisterHi")) {
081                         for (int k = j + 1; k <
082                             argToken.payload.size(); k++) {
083                             if
084                                 (argToken.payload.get(k).type.type_category.equals("RegisterHi")) {
085                                     regRangeOffset++;
086                                 }
087                             }
088                         }
089                     }
090                 }
091             }
092         }
093     }
094 }
```

```

070                     }
071             } else {
072                 if (!
073 (opCodeArgSub.is_entry_types.contains(argTokenSub.type_name))) {
074                     argFound = false;
075                     argFoundSub = false;
076                     break;
077                 }
078             } else {
079                 argFound = false;
080                 argFoundSub = false;
081                 break;
082             }
083         }
084
085         if (regRangeOffset > 0) {
086             //Register range offset is lower by 1 to allow for
RegisterRange type, -1 for list close
087             if ((argToken.payload.size() - regRangeOffset - 1) !=
= opCodeArg.sub_args.size()) {
088                 argFound = false;
089                 argFoundSub = false;
090                 break;
091             }
092         } else {
093             // -1 for group close
094             if ((argToken.payload.size() - 1) !=
opCodeArg.sub_args.size()) {
095                 argFound = false;
096                 argFoundSub = false;
097                 break;
098             }
099         }
100     }
101 } else {
102     argFound = false;
103     argFoundSub = false;
104     break;
105 }
106 }
107
108 if (argFound && !hasArgsSub && !argFoundSub) {

```

```

109         return opCode;
110     } else if (argFound && hasArgsSub && argFoundSub) {
111         return opCode;
112     }
113 }
114
115     throw new ExceptionNoOpCodeFound("Could not find an opCode that has
matching arguments for line number " + opCodeToken.lineNumAbs + " with
opCode '" + opCodeToken.source + "' at opCode argument index " +
opCodeArgIdx + " with sub argument index " + opCodeArgIdxSub);
116 }

001 public List<JsonObjIsOpCode> FindOpCodeMatches(String opCodeName, int
argLen) {
002     List<JsonObjIsOpCode> ret = new ArrayList<>();
003     for (JsonObjIsOpCode opCode : jsonObjIsOpCodes.is_op_codes) {
004         if (opCode.op_code_name.equals(opCodeName) && opCode.arg_len ==
argLen) {
005             ret.add(opCode);
006         }
007     }
008     return ret;
009 }

001 public int CountArgTokens(List<Token> payload, int argCount) {
002     return CountArgTokens(payload, argCount,
JsonObjIsEntryTypes.NAME_CAT_OPCODE, true);
003 }

001 public int CountArgTokens(List<Token> payload, int argCount, String
argCategory, boolean isOpCodeArg) {
002     for (Token token : payload) {
003         if (token.type != null && (JsonObjIsEntryType)
token.type.category.equals(argCategory)) {
004             argCount++;
005             if (isOpCodeArg) {
006                 token.isOpCodeArg = true;
007             } else {
008                 token.isDirectiveArg = true;
009             }
010         }
011
012         if (token.payload != null && token.payload.size() > 0) {

```

```

013         //List needs to be counted as an OpCode argument but Group
does not
014         if
(token.type_name.equals(JsonObjIsEntryTypes.NAME_START_LIST)) {
015             argCount++;
016             if (isOpCodeArg) {
017                 token.isOpCodeArg = true;
018             } else {
019                 token.isDirectiveArg = true;
020             }
021         } else {
022             argCount = CountArgTokens(token.payload, argCount,
argCategory, isOpCodeArg);
023         }
024     }
025 }
026     return argCount;
027 }
```

The second group of support methods to review from the AssemblerThumb class.

The second group of methods we have to review is shown in the previous listing. The first method we'll look at from this group is the `FindNextOpCodeLine` method. This support method is designed to look forward from the current line of tokenized data, `asmDataTokened`, and find the next line of assembly source code that contains an opcode or a directive `Token` and is not a label definition or an empty line. To handle this, on line 2, we check if there are any lines of source code left to look at. If so, on lines 3 – 8, we scan – forward looking – at the type of each `TokenLine` on line 5. If not, the method throws an exception describing the current error, line 10.

There's not much more to talk about with regard to this method so we'll move on. The second method in this group, `FindOpCodeArgMatches`, is similar to the `FindDirectiveArgMatches` method we've just reviewed, albeit a bit more complicated. Because opcodes use sub-tokens, a token that has a payload – like a comment, list, or group token – there's a fair amount of extra logic to support the second scan as we'll soon see. Let's take a look at some code!

On lines 2 – 10 the method variables are declared. Notice that there are a set of variables specifically for the sub-token scan. The method variables are temporary and used to track the opcode argument, the token the argument is associated with, the index of the argument, and if an argument match was found or not. The same search is performed for sub-tokens. We iterate over the provided list of matching opcodes, `opCodeMatches`, on line 12, and the loop variables are reset for the current loop iteration on lines 13 – 17.

Because opcodes are a little bit more complex than directives, we need to do some sorting. If you recall from our review of the ARM Thumb-1 instruction set, some opcodes have arguments listed out of order. Rather there is an order used to write the opcode, in source, and an order used for binary encoding of the opcode. These two orderings can differ and to be sure we have things prepared correctly we use the `JsonObjIsOpCodeArgSorter` class in its default configuration which sorts the opcode's arguments by argument index, as it appears in assembly source code, ascending as opposed to bit index, as it appears in the binary representation of the opcode, sorting.

To overcome this obstacle – a noted previously – we sort the arguments by the `arg_index` field in ascending order, line 20. Next, we iterate over the current opcode's arguments checking if we have a match to the provided list of arguments, `args`. Local variables `opCodeArg` and `opCodeArgIdx` are initialized based on the current argument we're checking, lines 23 – 24. If there is an `args` entry at the current inner loop index we set the `argToken` variable, line 27. If not, we exit the inner for loop, skipping this opcode's argument scan and move on to the next opcode in the `List` of opcodes to check.

If the opcode argument and provided argument token are defined, line 33, we move on to check the entry types associated with the current argument, lines 34 – 37. If the types do not match, we exit the inner for loop indicating that a match was not found. We check to see if the current argument has sub-arguments associated with it on line 39. By sub-arguments we mean `Tokens` of the argument entry type that are associated with a `Token`'s payload as opposed to a `TokenLine`'s payload. As noted earlier this occurs with opcode's that have groups or lists of arguments.

The code on lines 40 – 99 is used to iterate over the sub-arguments and make sure the known opcode arguments and the provided arguments have matching entry types. The code in this loop is very similar to the code used by the outer loop and in the method we reviewed in the previous support method group. As such, I'll let you review these lines of code on your own. Make sure you understand what's happening here before moving on. Note that there is special handling for register ranges, which only appear as sub-arguments in groups or lists, as well as offset tracking for the register ranges.

Register ranges are used with certain opcodes. I'll list an example of one here. In this case we have a list of registers are an opcode argument.

```
STMIA R0!, {R3, R4, R5, R6, R7}
```

In order to handle this we need to count the entries in the register range, {R3, R4, R5, R6, R7}, and increment the `regRangeOffset` variable. We also have to take into account the closing group or list token. You can see this circumstance handled by the support method on line 87, `((argToken.payload.size() - regRangeOffset - 1) != opCodeArg.sub_args.size())`, notice that we compare List sizes and take into account the register range and closing token.

A register range argument is used to take up the opcode's argument requirements for a series of registers of the same type. For example, if an opcode takes a list of registers as arguments you can satisfy this requirement with one register range argument, since the register range is used to specify a series of registers. Recall that we expand the compressed register range format, R0-R4, into explicit and individual tokens in step 7 of the assembly process. I'll list the step here,

**STEP 7:** Expand register ranges into individual register entries.  
Operative Method: `ExpandRegisterRangeTokens`

In this way the line of assembly source code, `PUSH {R0-R4, LR}`, becomes, `PUSH {R0, R1, R2, R3, R4, LR}`, in the assembler after step 7 is executed. That is why the method is designed to count the individual register entries and take them into account when comparing the length of the argument lists, as noted previously.

The remaining code on lines 108 – 112 is used to determine if a matching opcode, with all known arguments matching the arguments provided by type, has been found and returns it. There are two cases where we determine we have found a match and return the opcode object. The first is if the arguments have been found, `argFound == true`, and there are not sub-arguments to check, line 108. The second is if the arguments have been found and also the sub-arguments, line 110.

The next method up for review in this group is the `FindOpCodeMatches` method. This is a simple support method that's used to limit the number of matching opcodes by only finding those that have the same name and number of arguments. No further investigation into the argument types is done here. This is a quick preparation for a more rigorous comparison made later on. The find opcode match methods closely resemble the `FindDirectiveArgMatches`, and, `FindDirectiveMatches` methods we reviewed earlier in the chapter. Using the `FindOpCodeMatches` method we can quickly exclude the vast majority of available opcodes. On line 2 of the method, we initialize the return variable, `ret`. Since we are expecting multiple matches in some cases, we'll use a list to capture the results.

Next, on lines 3 – 7, we scan through the list of available opcodes, this data was loaded from the JSON files we reviewed in previous chapters, storing any that match on name and length of arguments. The resulting `List` of matches, if any, is returned on line 8 of the method. The following method in this group, `CountArgTokens`, is a pass through method and is only used to call the overloaded `CountArgTokens` method with some default arguments.

We'll take a look at the overloaded version of the `CountArgTokens` method now and omit the pass through from the review due to its simplicity. This method is designed to recursively count tokens that are of the argument entry type. The method takes a list of `Tokens` to scan, `payload`, an integer representing the current number of arguments tokens found thus far, `argCount`, a string description of the argument category, `argCategory`, and a Boolean flag indicating if this is an opcode or directive argument, `isOpCodeArg`.

The method starts off by iterating over the provided list of tokens, line 2. If the token is defined and has an entry type category that matches the method argument, `argCategory`, we increment the `argCount`, lines 3 and 4. We mark the token to indicate if it's an opcode or directive based argument on lines 5 – 9. On lines 12 – 24 we handle the recursive case, when there are sub-tokens. There is a special case here where we count a list start token as an opcode argument, but we do not count a group start token as an opcode argument. The group start token is more of a marker than anything else.

Next on lines 15 – 20 we handle incrementing the `argCount` variable and setting the token's Boolean flags to indicate an opcode or directive argument has been found. The recursive case is handled on line 22 where we call the method again with the token's payload, sub-arguments, as the list argument. That brings us to the conclusion of this group of support methods. We have two more groups to go over, which we'll review next. Take a moment to imagine how we might use these support methods, and the scans they perform, as we review them.

***Listing 17-9. AssemblerThumb.java Support Method Details 3***

```
01 public JsonObjectIsEntryType FindEntryType(String entryName) throws
ExceptionNoEntryFound {
02     for (JsonObjectIsEntryType entry : jsonObjIsEntryTypes.is_entry_types)
{
03         if (entry.type_name.equals(entryName)) {
04             return entry;
05         }
06     }
07     throw new ExceptionNoEntryFound("Could not find entry by name, '" +
entryName + "'", in loaded entry types.");
08 }

01 public String CleanRegisterRangeString(String range, String rangeDelim)
{
02     String ret = "";
03     for (char c : range.toCharArray()) {
04         if ((c + "").equals(rangeDelim) || Character.isDigit(c)) {
05             ret += c;
06         }
07     }
08     return ret;
```

```

09 }

01 public void CleanTokenSource(Token token) {
02     if (token != null && !Utils.IsEmpty(token.source)) {
03         token.source =
04             token.source.replace(JsonObjIsOpCode.DEFAULT_ARG_SEPARATOR, "");
05         token.source = token.source.replace(System.lineSeparator(), "");
06     }
07 }

01 public void LexerizeAssemblySource(int step) throws IOException {
02     if (eventHandler != null) {
03         eventHandler.LexerizeAssemblySourcePre(step, this);
04     }
05
06     Logger.wrl("AssemblerThumb: LoadAndLexAssemblySource: Lexerize
assembly source file");
07     LexerThumb lex = new LexerThumb();
08     asmDataLexed = lex.FileLexerize(asmDataSource);
09
10    if (eventHandler != null) {
11        eventHandler.LexerizeAssemblySourcePost(step, this);
12    }
13 }

01 public void TokenizeLexerArtifacts(int step) throws
ExceptionNoTokenerFound {
02     if (eventHandler != null) {
03         eventHandler.TokenizeLexerArtifactsPre(step, this);
04     }
05
06     Logger.wrl("AssemblerThumb: TokenizeLexerArtifacts");
07     TokenerThumb tok = new TokenerThumb();
08     asmDataTokened = tok.FileTokenize(asmDataLexed,
jsonObjIsEntryTypes);
09
10    if (eventHandler != null) {
11        eventHandler.TokenizeLexerArtifactsPost(step, this);
12    }
13 }

```

```

01 public int[]
FindValidLineEntryWithMatchingTokenType(JsonObjIsValidLine validLine,
Token token, int entry, int index) {
02     int count = 0;
03     for (JsonObjIsValidLineEntry validLineEntry :
validLine.is_valid_line) {
04         for (String validLineEntryType : validLineEntry.is_entry_types) {
05             if (token.type_name.equals(validLineEntryType)) {
06                 if (count >= entry) {
07                     return new int[]{count, validLineEntry.index};
08                 } else {
09                     break;
10                 }
11             }
12         }
13         count++;
14     }
15     return null;
16 }
```

The third group of support methods to review from the AssemblerThumb class.

In the group of support methods for us to review, listed previously, we have a few small methods to look over. The first method listed, `FindEntryType`, is used to search through the loaded instruction set's list of entry types to find one with a matching name. The method is very straightforward, there's not much more to discuss so we'll move onto the next method, `CleanRegisterRangeString`. This method is used to clean the characters used in a register range string. To do so we iterate over the characters stripping out everything except the range delimiting character and any digits. The main purpose of this support method is to simplify the information in a register range string removing and non-essential characters like the 'R' used to signify a register.

The next method in the group, `CleanTokenSource`, is also a simple method not too different from the one we just looked at. This method is used to clean some characters from the token's source string. The method clears out argument separator and line break characters, lines 2 – 3. The support method is called by the `ExpandRegisterRangeTokens` method and is used

to clear out characters that are unnecessary at this point in the assembly process and also it helps simplify the handling of register ranges. Following this method is the `LexerizeAssemblySource` method. This method is important because it's the first processing step done by the assembler and it converts the text-based assembly source code into `ArtifactLines` and `Artifacts`, as we've seen in previous reviews.

Notice that the method begins and ends with a call to the event handler, if defined, on lines 2 – 4 and 10 – 12. This is an example of the event callback method in action and is intended to be used to customize the assembly process. The actual work done in this method, lines 6 – 8, is very concise. We log the method execution, create a new `LexerThumb` instance, and call the class' `FileLexerize` method, storing the results in `AssemblerThumb` class' `asmDataLexed` field. The next method in the group follows a similar pattern.

The `TokenizeLexerArtifacts` method is also the next step in the data preparation process. Note that the method describes that it “tokenizes” the lexer's artifacts. This method has the same event callback checks and a very similar few lines of operative code, lines 6 – 8. A new instance of the `TokenerThumb` class is created on line 7, followed by a call to the class' `FileTokenize` method. Notice that this method takes a list of `ArtifactLines` and a set of entry types, loaded from a JSON data file, as arguments and returns a list of `TokenLines` with payloads that contain `Token` objects.

The last method in this group is the `FindValidLineEntryWithMatchingTokenType` method. This method is designed to be used as part of a progressive scan by the `ValidateTokenizedLine` method. The method takes a `JsonObjIsValidLine` object, `validLine`, as an argument. Along with the `token` argument that we're checking the valid line entry against and two index tracking arguments `entry` and `index`. Each `Token` in a `TokenLine` is passed into this method to verify that it matches the given `validLine`. On line 2 we initialize the only method variable, `count`. This variable is used to count the number of valid line entries we've looked at.

Next up, on line 3 we iterate over the valid line entries for the given valid line. Each valid line entry has an associated list of valid entry types. We iterate over this information starting on line 4. If the provided `token` is of the

correct type, line 5, and the index of the valid line entry, `count`, is greater than or equal to the `entry` argument then we have a match. On line 7 we return an array of integers with the index of the valid entry type, `count`, and the data driven `index` of the valid line entry as arguments. I'll post a snippet of the `is_valid_lines` JSON file here so you can get a better idea of the structure of the data we're looking at.

```
{  
    "obj_name": "is_valid_line",  
    "index": 2,  
    "empty_line": true,  
    "is_valid_line": [  
        {  
            "obj_name": "is_valid_line_entry",  
            "is_entry_types": ["Label"],  
            "index": 0  
        },  
        {  
            "obj_name": "is_valid_line_entry",  
            "is_entry_types": ["Comment"],  
            "index": 1  
        }  
    ]  
}
```

If no match is found the method returns a null value on line 15. This method is designed to work through subsequent calls using the `entry` argument to move where the pertinent comparison occurs. This is to prevent matching on the first, or previous entry type, if such a match exists. We'll see this method in use when we encounter it in the next section. We have three more methods to review and they'll constitute the last section of support methods we'll cover. Due to the complexity of some of these methods I'll have to break them up into multiple listings. We'll start things off by looking into the `ProcessShift` method.

***Listing 17-10. AssemblerThumb.java Support Method Details 4***

```

01 public Object[] ProcessShift(Integer tIntIn, JsonObjectNumRange num_range,
JsonObjectBitShift bit_shift, JsonObjectBitSeries bit_series, String source, int
lineNumAbs, boolean bitLen2x, boolean checkRange) throws
ExceptionNumberInvalidShift, ExceptionNumberOutOfRange {
02     Integer tInt = tIntIn;
03     String resTmp1 = null;
04     Integer t = null;
05
06     if (num_range != null && num_range.ones_complement == true) {
07         tInt = ~tInt;
08     }
09
10    resTmp1 = Integer.toBinaryString(tInt);
11    Integer bltInt = tInt;
12    Integer tInt2 = tInt;
13    String blResTmp1 = resTmp1;
14    if (bit_shift != null) {
15        if (bit_shift.shift_amount > 0) {
16            if (!Utils.IsEmpty(bit_shift.shift_dir) &&
bit_shift.shift_dir.equals(NUMBER_SHIFT_NAME_LEFT)) {
17                resTmp1 = Utils.ShiftBinStr(resTmp1,
bit_shift.shift_amount, false, true);
18            } else if (!Utils.IsEmpty(bit_shift.shift_dir) &&
bit_shift.shift_dir.equals(NUMBER_SHIFT_NAME_RIGHT)) {
19                resTmp1 = Utils.ShiftBinStr(resTmp1,
bit_shift.shift_amount, true, true);
20            } else {
21                throw new ExceptionNumberInvalidShift("Invalid number shift
found for source '" + source + "' with line number " + lineNumAbs);
22            }
23
24            t = Integer.parseInt(resTmp1, 2);
25            if (tInt2 % 4 == 2 && bit_shift.shift_amount == 2 &&
bit_shift.shift_dir.equals(NUMBER_SHIFT_NAME_RIGHT) == true) {
26                t++;
27                resTmp1 = Integer.toBinaryString(t);
28            }
29        }
30    }
31
32    if (bitLen2x) {
33        resTmp1 = Utils.FormatBinString(resTmp1, bit_series.bit_len * 2);
34    } else {

```

```

35     resTmp1 = Utils.FormatBinString(resTmp1, bit_series.bit_len);
36 }
37 tInt = Integer.parseInt(resTmp1, 2);
38
39 if (num_range != null && checkRange) {
40     if (tInt < num_range.min_value || tInt > num_range.max_value) {
41         throw new ExceptionNumberOutOfRange("Integer value " + tInt +
" is outside of the specified range " + num_range.min_value + " to " +
num_range.max_value + " for source '" + source + "' with line number " +
lineNumAbs);
42     }
43 }
44
45 return new Object[]{resTmp1, tInt, blResTmp1, bltInt};
46 }

```

The first of the remaining three support methods left to review from the AssemblerThumb class.

The `ProcessShift` method is used to apply a bit shift to an integer. Bit shifting happens in a few opcodes where an integer argument has a bit shift applied to it. The `ProcessShift` method takes a number of arguments let's go over them now. The first argument is the integer `tIntIn`. This is the number we'll be applying the bit shift to. The method also takes a `num_range` to provide information about the valid range for the given integer. The subsequent argument, `bit_shift`, provides information about how to shift the bits of the given integer while the `bit_series` argument is used to define the required bit length of the shifted integer value.

Following these method arguments is a string representation of the assembly source code, `source`, this is used in exceptions generated by this method. The `lineNumAbs` argument is an integer representing the absolute line number this shift operation is associated with. The remaining arguments are Boolean flags. The first one, `bitLen2x`, is used to indicate the resulting shifted numeric value should have a padded bit length that is two times the length indicated by the `bit_series` argument. The last Boolean, `checkRange`, is used to indicate the resulting, shifted, numeric value should have its range checked to make sure it isn't larger or smaller than expected.

Take a moment to notice that this method throws an `ExceptionNumberInvalidShift` or an `ExceptionNumberOutOfRange` exception in certain cases. Local method variables are initialized on line 2 – 4. The integer `tInt` is used to hold a copy of the `tIntIn` argument while the variables string `resTmp1` and integer `t` are used as temporary variables. On lines 6 – 8 we check to see if we have a valid `num_range` object and if a one's complement is required. If so we use Java bit operator, “~”, and apply it to the `tInt` variable.

Now that we're ready to do some bit shifting we get a binary representation of the `tInt` variable in string form and store it in the `resTmp1` method variable, line 10. Then on lines 11 – 13 we initialize three more method variables, `bltInt`, `tInt2`, and `blResTmp1`. Notice that these variables are used to hold copies of the `tInt` and `resTmp1` method variables before we perform any bit shifting. The block of code responsible for performing the shift starts on line 14 and continues to line 30. If the `bit_shift` argument is defined and if the `shift_amount` is greater than zero we begin to apply the shift. Due to convenience we are going to sometimes work with numeric values and in other cases we're going to work with string representations of binary numbers.

On lines 16 – 22 we use the string representation of a binary number, `resTmp1`, which is generated from the `tInt` method variable. The shifts are applied using the utility method `ShiftBinStr` on lines 17 and lines 19. If no shift direction was found in the `bit_shift` argument an exception is thrown indicating the issue, line 21. The next operation we want to perform on the shifted numeric value occurs on lines 24 – 28. The shifted binary string is converted back into an integer value on line 24.

The next snippet of code, lines 25 – 28 is a bit of a hack. Certain required shifts of the ARM Thumb-1 instruction set have to be word aligned, divisible by four, and this is the code the handles it. Although I'm not in love with the way I implemented it. To explain further let me show an example of the ADD opcode that requires the shift I just mentioned. The fist example shows a word aligned value. Notice that the end result is the number 26.

```
OpCode:      ADD    SP, #-104
Before Shift: 00000000 01101000
Integer Value: 104
```

```

Desc:          word aligned bits 1,0 = 0
After Shift:   00000000 00011010
Integer Value: 26

OpCode:        ADD    SP, #-102
Before Shift:  00000000 01100110
Integer Value: 102
Desc:          non-word aligned bits 1,0 != 0
After Shift:   00000000 00011001
Integer Value: 25
Adjustment:    00000000 00011010
Desc:          increment by 1
Integer Value: 26

```

In order to maintain a word aligned value as the opcode requires we could adjust the original number value, 102, or we could adjust the value after the shift is applied. Incrementing the shifted value by one will make the necessary adjustment and as you can see in the previous example the resulting values are now the same, 26. The code in the snippet checks for the proper word alignment issue, if (`tInt % 4 == 2`) and the bit shift amount is 2, and the direction is a right shift. This will trigger our little hack snippet of code.

There is one other caveat I'd like to mention. While the GenAsm assembler handles numeric ranges using the `JsonObjNumRange` class, it doesn't have the ability to exclude numbers from the range. The same ADD opcode that we just discussed doesn't support values less than 4, so -3 – 3 aren't considered valid values. If you think about the required 2 bit right shift you'll understand why. Simply put those values will be lost in the shift. Our assembler won't care and as such will generate incorrect machine code so you must take care to use the correct values with this particular ADD opcode, there are a few different ADD opcodes so be aware of the context.

```
;genasm - doesn't throw an error but isn't supported in hardware
10110000 10000001 0xB0 81           ADD    SP, #-2
```

```
;vasm - illegal unsupported out of range opcode
ADD    SP, -2           ;vAsm: "Illegal out of range" ARMThumb_Sim Asm:
B0FC, 1011000011111100 ShellStorm Asm: B0FF, 10110000
```

The assembler output, listed previously, show the VASM assembler catching the incorrect numeric argument to the ADD opcode while our assembler just spits out a binary representation without a care. Again, just be careful using this version of the ADD opcode and you'll be fine. Getting back to the remainder of the method we're reviewing. The code on lines 32 – 36 is responsible for formatting the binary string representation of the shifted number. If the `bitLen2x` Boolean flag is true then the binary string is formatted with twice the length.

On line 37 the method variable `tInt` is updated based on the latest binary representation. The last block of code, lines 39 – 43, is responsible for checking the range of the `tInt` variable, if the `checkRange` Boolean flag is true. The last line of the method, line 45, is where we return an array of objects summarizing the shift operation. The first two values are the new binary string representation of the shifted numeric value, `resTmp1`, and an integer representation of the same value, `tInt`. We also send across the original binary string representation and integer value represented by the `blResTmp1` and `bltInt` variables included in the array.

In the next listing we'll take a look at the `ProcessSymbolValue` method. This is the second to last support method up for our review. This method is used to return a value for the specified label, aka symbol, taking into account the associated label reference character. Let's take a look.

#### *[Listing 17-11. AssemblerThumb.java Support Method Details 5](#)*

```
01 public Object[] ProcessSymbolValue(String source, JsonObjectBitSeries
bit_series, int lineNumberAbs, TokenLine line) throws ExceptionNoSymbolFound,
ExceptionNoSymbolValueFound {
02     char c = source.charAt(0);
03     String label = source.substring(1);
04     Symbol sym = symbols.symbols.get(label);
05     String resTmp1 = null;
06
07     if (sym != null) {
08         resTmp1 =
Utils.FormatBinString(Integer.toBinaryString(sym.addressInt),
bit_series.bit_len);
```

```

09     } else {
10         throw new ExceptionNoSymbolFound("Could not find symbol for label
11         '" + label + "' with line number " + lineNumAbs);
12     }
13     Integer tInt = null;
14     if (c == JsonObjIsEntryTypes.NAME_LABEL_REF_START_ADDRESS) {
15         //label address =
16         tInt = sym.addressInt;
17         Logger.wrl("Symbol lookup: Found start address, '" + tInt + "'",
for symbol, '" + label + "'.");
18
19     } else if (c == JsonObjIsEntryTypes.NAME_LABEL_REF_START_VALUE) {
20         //label value ~
21         if (sym.value != null && sym.isStaticValue) {
22             tInt = sym.value;
23             Logger.wrl("Symbol lookup: Found start value, '" + tInt + "'",
for symbol, '" + label + "'.");
24         } else {
25             throw new ExceptionNoSymbolValueFound("Could not find symbol
value for label '" + label + "' with line number " + lineNumAbs);
26         }
27
28     } else if (c == JsonObjIsEntryTypes.NAME_LABEL_REF_START_OFFSET) {
29         //label address offset -
30         if (line.addressInt < sym.addressInt) {
31             tInt = (sym.addressInt - line.addressInt) +
jsonObjIsOpCodes.pc_prefetch_bytes;
32         } else {
33             tInt = (line.addressInt - sym.addressInt) -
jsonObjIsOpCodes.pc_prefetch_bytes;
34         }
35         Logger.wrl("Symbol lookup: Found REF start offset, '" + tInt + "'",
for symbol, '" + label + "'.");
36
37     } else if (c ==
JsonObjIsEntryTypes.NAME_LABEL_REF_START_OFFSET_LESS_PREFETCH) {
38         //label address offset minus prefetch `
39         if (line.addressInt < sym.addressInt) {
40             tInt = ((sym.addressInt - line.addressInt) -
jsonObjIsOpCodes.pc_prefetch_bytes);
41         } else {

```

```

42         tInt = -1 * ((line.addressInt - sym.addressInt) +
json0bjIsOpCodes.pc_prefetch_bytes);
43     }
44     Logger.wrl("Symbol lookup: Found REF start offset less pre-fetch,
'" + tInt + "', for symbol, '" + label + "'.");
45
46 } else {
47     throw new ExceptionNoSymbolFound("Could not find symbol for label
'" + label + "' with line number " + lineNumAbs + " and label prefix " +
c);
48 }
49
50 return new Object[]{tInt, label};
51 }

```

The second of the remaining three support methods left to review from the AssemblerThumb class.

Before we jump into the method review let's refresh our memory regarding the label reference character. I'll list the static fields referenced by this method along with some comments so we can clearly see what they mean.

```

/**
 * A static string that represents the label reference
 * address character. Used to return the address of the
 * label referenced.
 */
public static char NAME_LABEL_REF_START_ADDRESS = '=';

/**
 * A static string that represents the label reference
 * value character. Used to return the value, if available,
 * of the label referenced.
 */
public static char NAME_LABEL_REF_START_VALUE = '~';

/**
 * A static string that represents the label address offset,
 * from the current line, ignoring any pre-fetch values.

```

```

*/
public static char NAME_LABEL_REF_START_OFFSET = '-';

/**
 * A static string that represents the label address offset,
 * from the current line, taking into account the pre-fetch.
 */
public static char NAME_LABEL_REF_START_OFFSET_LESS_PREFETCH = ``';

```

Take a moment to read over the entries above and we'll see how they are expressed in code. Let's start things off by reviewing the method's arguments. The first argument, `source`, is a string value that is the assembly source code containing the label reference character and the label name. The second argument, `bit_series`, provides a little information about the binary string representation of the symbol's address. The next argument, `lineNumAbs`, is mainly used in logging and error reporting so the user can see what line number the source string is from. Lastly, the `line` argument, which is an instance of the `TokenLine` class is used to provide more information about the line the source code is on.

The method variables are initialized on lines 2 – 5. The first variable, `c`, is used to hold the symbol reference character. It is set to the first character of the symbol reference assembly source code, line 2. The next variable, `label`, is set to the remainder of the `source` string. The symbol we're working is pulled from the `AssemblerThumb` class' `symbol` field which is a repository for all symbols defined in the current assembly source code. The retrieved symbol is stored in the `sym` variable on line 4. The last method variable initialized is the `resTmp1` variable and it is used to verify the retrieved symbol's address.

As noted earlier we run a quick check to make sure the `sym` variable is valid, lines 7 – 11. If the variable `sym` is not null we create a binary string representing the symbol's address to verify that we have a valid address. The value we generate here isn't really used. Note we throw an exception on line 10 if the retrieved symbol is null. A convenience variable is declared on line 13, it is used to simplify the logging statements making them more uniform. The block of code on lines 14 – 48 is a large if-else statement that handles processing the different symbol reference characters. On line 14 we check to

see if the symbol reference character is “=”. If so we set `tInt` equal to the address of the symbol in the assembly source code and logs the occurrence.

On line 19 we check for the second symbol reference character, “~”, which is used to pull the value assigned to a symbol, if any. On line 21 we check to make sure the symbol’s value is not null and that the symbol is of the correct type, (`sym.isStaticValue == true`). Note that on line 25 we throw an `ExceptionNoSymbolValueFound` exception if the symbol’s value doesn’t pass our checks. Again we use the `tInt` variable but this time we set it equal to the symbol’s value and log the occurrence.

Next up, the block of code on lines 28 – 37 is there to handle the “–” symbol reference character. This character signifies that we want the offset to the specified label’s address not taking into account any prefetch that is done the hardware. We want to figure out if the symbol’s address is greater than or less than the line of source code that references the symbol, `line`. A quick glance at the code on lines 30 – 34 seems to indicate that we do consider the prefetch when calculating the address offset. Let me explain what’s happening here.

In this case we aren’t concerned with the prefetch in the sense that we allow it to exist in our calculated address offset. Not in the sense that we’re removing it from the offset. It’s a bit counter intuitive. Let’s talk our way through it. If the source code line’s address is less than the symbol’s address we calculate the difference, (`sym.addressInt - line.addressInt`), and then add in the prefetch value. Going in the opposite direction we subtract the prefetch value. This symbol reference character isn’t used much but I kept it in place for debugging purposes and comparison to the next character which we’ll review now.

The last symbol reference character handles by this method is the offset minus prefetch character, “`”. This symbol reference character works opposite of the character we just reviewed. In this case we subtract the prefetch if the symbol’s address is greater than the line’s address and add the prefetch to the difference otherwise, lines 39 – 43. Similar to other clauses we log the occurrence on line 44. If no symbol matching the given label is found we throw an `ExceptionNoSymbolFound` on line 47 and to close out the method we return an array of objects containing the value found, `tInt`, and

the label used, line 50. That brings us to the conclusion of the ProcessSymbolValue method review.

We have one more support method left to look at to complete our review of the AssemblerThumb class' support methods, the BuildBinOpCodePrep method. This method is used at the final stage of the assembly process where we start building up binary representations of the assembly source code. This method is a long one so we'll break it up into more manageable pieces. Let's take a look.

*Listing 17-11. AssemblerThumb.java Support Method Details 6*

```
001 public void BuildBinOpCodePrep(int step, TokenLine line) throws
ExceptionOpCodeAsArgument, ExceptionUnexpectedTokenWithSubArguments {
002     if (!line.isEmpty && !line.isLineDirective &&
line.isLineOpCode) {
003         JsonObjIsOpCode opCode = line.matchesOpCode.get(0);
004         List<JsonObjIsOpCodeArg> opCodeArgs = opCode.args;
005         List<BuildOpCodeThumb> buildEntries = new ArrayList<>();
006         List<JsonObjIsOpCodeArg> opCodeArgsSub = null;
007         Collections.sort(opCodeArgs, new
JsonObjIsOpCodeArgSorter(JsonObjIsOpCodeArgSorter.JsonObjIsOpCodeArgSorter
Type.ARG_INDEX_ASC));
008         Collections.sort(line.payload, new
TokenSorter(TokenSorterType.INDEX_ASC));
009         BuildOpCodeThumb tmpB = null;
010         int opCodeArgIdx = 0;
011         int lOpCodeArgIdx = 0;
012
013         //Prepare BuildOpCodeEntry list based on tokens and json
arguments
014         for (Token token : line.payload) {
015             lastToken = token;
016             if (token.isOpCodeArg) {
017                 tmpB = new BuildOpCodeThumb();
018                 tmpB.isOpCodeArg = true;
019                 tmpB.opCodeArg = (JsonObjIsOpCodeArg)
opCodeArgs.get(opCodeArgIdx);
020                 tmpB.bitSeries = tmpB.opCodeArg.bit_series;
021                 tmpB.tokenOpCodeArg = token;
022 }
```

```
023         if (tmpB.opCodeArg.bit_index != -1) {  
024             buildEntries.add(tmpB);  
025         }  
026  
027         opCodeArgIdx++;  
028 ...
```

Part one of the last support method to review, `BuildBinOpCodePrep`, from the `AssemblerThumb` class.

As noted in the method's code comments the `BuildOpCodePrep` method is used to prepare the `BuildOpCodeThumb` objects by connecting the given token line's tokens with instances of the holder classes as needed. We'll get a better idea of how it all works once we start reviewing the code so let's jump to it. Let's review the methods arguments first there are only two for us to go over. The fist argument, `step`, isn't currently used but we keep it around in any case. The `step` argument comes from the main processing method of the `AssemblerThumb` class.

The second argument is an instance of the `TokenLine` class, `line`, and is the basis for the work this method has to do. As you can tell this method is called for each token line of assembly source we need to process. Also note that this method throws `ExceptionOpCodeAsArgument` and `ExceptionUnexpectedTokenWithSubArguments` exceptions. On line 2 we check to make sure the given `line` argument is of the correct type before continuing. If the `line` is not empty, is not a directive line, and is explicitly marked as an opcode line we can proceed an initialize some method variables.

The first method variable, `opCode`, is an instance of the `JsonObjIsOpCode` class, line 3, and is initialized to the first matching `JsonObjIsOpCode` instance in the `matchesOpCode` field of the `TokenLine` class. This opcode matches the signature of the opcode and arguments contained in the `TokenLine`. The next method variable, `opCodeArgs`, is set to reference the `opCode`'s arguments on line 4. Now we have quick access to the opcode and its arguments. On line 5 we initialize a new method variable, `buildEntries`, as a List of `BuildOpCodeThumb` objects.

The `BuildOpCodeThumb` class holds all the pertinent objects and data needed to convert a line of assembly source code to a binary representation including the binary representation itself in big- and little-endian format. Similar to the `opCodeArgs` variable we have an `opCodeArgsSub` variable declared on line 6 to hold information about sub-arguments, think groups and lists, if any. The next two lines of code are responsible for sorting the `opCodeArgs` List by index, line 7, and the line's payload, line 8, by index ascending. This is the sort order required for the work we have to do here. The next three variables, 9 – 11, are temporary variables used in the method's processing loop. Which we'll take a look at next.

We iterate over the line's payload and work with each `token` in it starting on line 14. The class field, `lastToken`, is updated to mark what token was being worked on last. The first block of code, lines 16 – 106 handles tokens that are opcode arguments. This is a large snippet of code so we'll have to break it up into more than one listing. The `tmpB` variable is set to a new instance of the `BuildOpCodeThumb` class on line 17. The `isOpCodeArg` Boolean flag is set to true and a few of the class' fields are set on line 19 – 21. Notice that we set the `opCodeArg` field of the `tmpB` variable to the current argument tracked by the `opCodeArgIdx` variable.

We need to make sure the argument we're working on is one that will end up in the final binary representation of the opcode, lines 23 – 25. We have to perform this check because some arguments that appear in the assembly source code are not present in the binary representation. These tokens are marked by the opcode arguments `bit_index` being set to a -1. If the `bit_index` is not -1 we add the argument to the `buildEntries` list, line 24, and increment the `opCodeArgIdx` variable on line 27. Remember we're working with opcode arguments, the opcode itself will be handled a little later on. But, before we get to that part of the method we need to process this token's sub-arguments, if any.

#### *Listing 17-12. AssemblerThumb.java Support Method Details 7*

```
029 if (!Utils.IsEmpty(token.payload) && !  
Utils.IsEmpty(tmpB.opCodeArg.sub_args)) {  
030     if (token.type_name.equals(JsonObjIsEntryTypes.NAME_START_LIST)) {  
031         10pCodeArgIdx = 0;  
032         JsonObjIsOpCodeArg opCodeArgList = tmpB.opCodeArg;  
033         opCodeArgsSub = tmpB.opCodeArg.sub_args;
```

```

034     Collections.sort(opCodeArgsSub, new
JsonObjIsOpCodeArgSorter(JsonObjIsOpCodeArgSorter.JsonObjIsOpCodeArgSorter
Type.ARG_INDEX_ASC));
035     Collections.sort(token.payload, new
TokenSorter(TokenSorterType.INDEX_ASC));
036     boolean regRangeLow = false;
037     boolean regRangeHi = false;
038
039     if
(opCodeArgsSub.get(0).is_entry_types.contains(JsonObjIsEntryTypes.NAME_REG
ISTER_RANGE_LOW)) {
040         regRangeLow = true;
041     } else if
(opCodeArgsSub.get(0).is_entry_types.contains(JsonObjIsEntryTypes.NAME_REG
ISTER_RANGE_HI)) {
042         regRangeHi = true;
043     }
044
045     for (Token ltoken : token.payload) {
046         if (ltoken.isOpCodeArg) {
047             if (regRangeLow && !
ltoken.type_name.equals(JsonObjIsEntryTypes.NAME_REGISTER_LOW)) {
048                 lOpCodeArgIdx++;
049             } else if (regRangeHi && !
ltoken.type_name.equals(JsonObjIsEntryTypes.NAME_REGISTER_HI)) {
050                 lOpCodeArgIdx++;
051             }
052
053             tmpB = new BuildOpCodeThumb();
054             tmpB.isOpCodeArgList = true;
055
056             if
(ltoken.type_name.equals(JsonObjIsEntryTypes.NAME_STOP_LIST)) {
057                 tmpB.opCodeArgList = opCodeArgList;
058             } else {
059                 tmpB.opCodeArgList = (JsonObjIsOpCodeArg)
opCodeArgsSub.get(lOpCodeArgIdx);
060             }
061
062             tmpB.bitSeries = tmpB.opCodeArgList.bit_series;
063             tmpB.tokenOpCodeArgList = ltoken;
064
065             if (tmpB.opCodeArgList.bit_index != -1) {

```

```

066         buildEntries.add(tmpB);
067     }
068
069     } else if (ltoken.isOpCode) {
070         throw new ExceptionOpCodeAsArgument("Found OpCode token
entry where a sub-argument should be on line " + line.lineNumAbs + " with
argument index " + ltoken.index + " and parent argument index " +
token.index);
071     }
072 }
073 } ...

```

Part two of the last support method to review, BuildBinOpCodePrep, from the AssemblerThumb class.

Now that we've processed the opcode argument we need to check if this argument has any sub-arguments. To verify, we have to check that the token has a payload and that the opCodeArg has sub\_args, line 29. Once we've verified that we have sub-arguments to process we need to check if we're dealing with a list or a group. First we check if we have a list to process, line 30, by testing the type\_name of the token. If so we have a few variables to prep, lines 31 – 37. The lOpCodeArgIdx method variable is set to 0 on line 31. Next, we declare a local variable, opCodeArgList, and set it equal to the current opcode argument, opCodeArg, line 32. We set the variable, opCodeArgSub, to the sub\_args field of the current opcode argument, opCodeArg.

Now that we have quick references to pertinent object instances in place we can prepare the lists, sub\_args and payload, by sorting them in index ascending order. This is similar to the preparation we performed on the opcode argument lists earlier. Two local variables, Boolean flags, are initialized on lines 36 – 37. These are used to track if we are working with a register range. The reason why we can be sure that the register range, if we're dealing with one, occurs in the first position of the sub arguments is because lists are always lists of registers in the ARM Thumb-1 instruction set. On lines 39 – 43 we check if we're dealing with a register range and set the appropriate Boolean flag to true.

On line 45 we iterate over the current token's payload, remember these are sub-arguments. If the current token is an opcode argument we process to process it, lines 47 – 68. List sub-arguments are members of the list start token's payload. The list opcode argument is stored in the `opCodeArgList` local variable, on line 32, and this will come in handy soon. Keep in mind this is the token that has all the list sub-tokens. If we have come across a register range we check to see if the current sub-token is of the expected register type, 47 – 51. We perform this check for both low and hi registers. If not we increment the `1OpCodeArgIdx` variable.

A new instance of the `BuildOpCodeThumb` class is initialized on line 53 and stored in the temporary variable `tmpB`. Note that we set the `isOpCodeArgList` to true to indicate that this object is part of a list, line 54. If we've encountered the stop list sub-argument we set the `opCodeArgList` field of the `tmpB` variable to the parent token, line 57. If not we pull out the sub-argument and use that reference instead, 59. The `bit_series` field and a reference to the sub-argument token, `ltoken`, are set on lines 62 – 63. If the `bit_index` is not -1 then we add the `tmpB` variable into the `buildEntries` list. The code on lines 69 – 71 is there to throw an exception if the sub-argument token we're processing is not an opcode argument but an actual opcode. Since this should never happen we throw an exception if it's encountered.

The code on lines 73 – 101 is very similar to the block of code we just reviewed. Groups and lists are handled in almost the same way in most cases. Don't forget that the group or list start token contains the sub-arguments including the group or list stop token. We don't really care about the stop token. It is ignored most of the time because it doesn't end up in the binary representation of an opcode and we can tell when we got to the end of a sub-arguments list without it. On lines 102 – 104 we throw an exception if the sub-argument token isn't a list or group token because this case should not exist.

Lastly, the snippet of code on lines 106 – 113 is responsible for creating a `BuildOpCodeThumb` instance that represents the opcode. Take a moment to look it over and make sure you understand it before moving on. It's similar and simpler than the code required to process argument and sub-argument tokens. The final line of code in the method, line 116, sets the `buildEntries` field of the `TokenLine` class, line, so we have everything

ready for when we start building out the binary representation of each line of assembly code. That brings us to the conclusion of the support method review of the `AssemblerThumb` class. In the next section we'll take a look at the class' all important main methods.

## Chapter Conclusion

In this chapter we started the herculean task of reviewing the `AssemblerThumb` class. Well maybe not quite herculean but certainly no walk in the park. We covered a few key aspects of the class review here but, due to the size of the class and the depth of its responsibilities we have to split this discussion across a few chapters. Let's review what we've covered thus far.

- Listing of the assembler's general steps, with a brief description, including listing the operative main method associated with each step.
- Review of the class' static class fields.
- Review of the class' regular class fields.
- Review of the class' pertinent method outline and class declaration.
- Review of the class' support methods.

That's not too shabby, we've finished reviewing every aspect of the class I wanted to talk about except for the main methods. There's a lot of code left for us to cover so we've split the discussion of the remaining main methods into the next two chapters, Chapters 18 and 19. There's a lot of code left for us to review but, don't worry we'll get through it.

# Chapter 18: The Assembler Part 2

Welcome to part 2 of a 3 part assembler review. In part 1, Chapter 17, we managed to review the overall process listing each step and its associated operative main method, the class fields, headers, pertinent class methods, and the `AssemblerThumb` class' support methods. No small task. In part 2 we'll start reviewing the class' main methods.

These main methods are where all the work is done and as such, they will be long and somewhat challenging methods to review. It will take us two chapters to review all 13 steps the assembler performs, including the associated operative main methods. We'll start this review here, in Chapter 18 and complete it in Chapter 19. I'll list the main methods we'll be looking at now.

*Listing 18-1. AssemblerThumb.java Main Method Outline 1*

```
//Main Methods
public void RunAssembler(JsonObjIsSet jsonIsSet, String
assemblySourceFile, List<String> assemblySource, String outputDir, Object
otherObj, AssemblerEventHandler asmEventHandler, boolean verbose, boolean
quellOutput);

public void LoadAndParseJsonObjData(int step);
```

```
public void LinkJsonObjData(int step);
public boolean ValidateTokenizedLines(int step);
public boolean ValidateTokenizedLine(int step, TokenLine line,
JsonObjIsValidLines validLines, jsonObjIsValidLine validLineEmpty);

public void CollapseCommentTokens(int step);
public void ExpandRegisterRangeTokens(int step);
public void CollapseListAndGroupTokens(int step);
public void PopulateOpCodeAndArgData(int step);
public void PopulateDirectiveArgAndAreaData(int step);

public void ValidateOpCodeLines(int step);
public void ValidateDirectiveLines(int step);
public void CleanAreas();
public void CleanSymbols();
public void BuildBinLines(int step, List<TokenLine> areaLines, AreaThumb
area);

public void BuildBinOpCodePrep(int step, TokenLine line);
public void BuildBinDirective(int step, TokenLine line);
public void BuildBinOpCode(int step, TokenLine line);
```

A listing of the AssemblerThumb class' pertinent main and support methods.

Let's pick things up where we left off and start the class' main method review.

## Main Method Details: AssemblerThumb.java

The AssemblerThumb class has a number of important main methods for us to review. We'll start the review process by looking at the RunAssembler method. This method is long, so we'll review it in parts. It's also the main execution point for this class and will serve as a reference to the order of the remaining main methods we'll review. We'll refer back to the steps outlined in this method, and mentioned in Chapter 17, as we work through the class' main method review.

*Listing 18-2. AssemblerThumb.java Main Method Details 1*

```

001 public void RunAssembler(JsonObjIsSet jsonIsSet, String
assemblySourceFile, List<String> assemblySource, String outputDir, Object
otherObj, AssemblerEventHandler asmEventHandler, boolean verbose, boolean
quellOutput) throws Exception {
002     try {
003         verboseLogs = verbose;
004         quellFileOutput = quellOutput;
005         eventHandler = (AssemblerEventHandlerThumb) asmEventHandler;
006         if (eventHandler != null) {
007             eventHandler.RunAssemblerPre(this, jsonIsSet,
assemblySourceFile, assemblySource, outputDir, otherObj);
008         }
009
010         Logger.wrl("AssemblerThumb: RunAssembler");
011         other = otherObj;
012         rootOutputDir = outputDir;
013         asmStartLineNumber = 0;
014
015         jsonSource = new Hashtable<String, String>();
016         isaLoader = new Hashtable<String, Loader>();
017         isaData = new Hashtable<String, JsonObj>();
018         isaDataSet = jsonIsSet;
019         asmSourceFile = assemblySourceFile;
020         asmDataSource = assemblySource;
021         symbols = new Symbols();
022
023         requiredDirectives = new ArrayList<String>();
024         requiredDirectives.add("@AREA");
025         requiredDirectives.add("@TTL");
026         requiredDirectives.add("@ENTRY");
027         requiredDirectives.add("@END");
028         ...

```

The AssemblerThumb class' RunAssembler method, part 1.

The first block of code, listed previously, is responsible for initializing some important class fields used throughout the assembly process. We've seen this method signature before when we reviewed the project's base classes, so I won't cover it again here. On line 2 we have the start of an important try – catch statement. This try – catch surrounds all of the

`RunAssembler` method's code and is the center point for catching and reporting exceptions.

Next, on lines 3 – 5, we store some method arguments in class fields including the `eventHandler` field on line 5. If the `eventHandler` field is defined, line 6, we call the `RunAssemblerPre` method to handle any customizations. We log that this method is executing and initialize a few more class fields on lines 10 – 13. The three class fields that are used to track the source, loader, and holder classes used by the assembler are initialized on lines 15 – 17.

The `isDataSet` field is set on line 18 with the provided instruction set object. Following this, the `asmSourceFile` and `asmDataSource` fields are set, lines 19 and 20, respectively. The `symbols` field is initialized on line 21. Lastly, on lines 23 – 27 we initialize the required directives list and populate it. We'll continue the review of this method in the next listing and pick things up at step one.

#### *`Listing 18-3. AssemblerThumb.java Main Method Details 2`*

```
029 Logger.wrl("");
030 lastStep = 1;
031 Logger.wrl("STEP 1: Process JsonObjIsSet's file entries and load then
parse the json object data");
032 if (eventHandler != null) {
033     eventHandler.RunAssemblerPreStep(lastStep, this);
034 }
035 LoadAndParseJsonObjData(lastStep);
036
037 Logger.wrl("");
038 lastStep = 2;
039 Logger.wrl("STEP 2: Link loaded json object data");
040 if (eventHandler != null) {
041     eventHandler.RunAssemblerPreStep(lastStep, this);
042 }
043 LinkJsonObjData(lastStep);
044
045 Logger.wrl("");
046 lastStep = 3;
047 Logger.wrl("STEP 3: Load and lexerize the assembly source file");
048 if (eventHandler != null) {
```

```

049     eventHandler.RunAssemblerPreStep(lastStep, this);
050 }
051 LexerizeAssemblySource(lastStep);
052 if (!quellFileOutput) {
053     Utils.WriteObject(asmDataLexed, "Assembly Lexerized Data",
054         "output_lexer.json", rootOutputDir);
055 }
056 Logger.wrl("");
057 lastStep = 4;
058 Logger.wrl("STEP 4: Tokenize the lexerized artifacts");
059 if (eventHandler != null) {
060     eventHandler.RunAssemblerPreStep(lastStep, this);
061 }
062 TokenizeLexerArtifacts(lastStep);
063 if (!quellFileOutput) {
064     Utils.WriteObject(asmDataTokened, "Assembly Tokenized Data",
065         "output_tokened_phase0_tokenized.json", rootOutputDir);
066 }
067 Logger.wrl("");
068 lastStep = 5;
069 Logger.wrl("STEP 5: Validate token lines");
070 if (eventHandler != null) {
071     eventHandler.RunAssemblerPreStep(lastStep, this);
072 }
073 ValidateTokenizedLines(lastStep);
074 if (!quellFileOutput) {
075     Utils.WriteObject(asmDataTokened, "Assembly Tokenized Data",
076         "output_tokened_phase1_valid_lines.json", rootOutputDir);
077 }

```

The AssemblerThumb class' RunAssembler method, part 2.

Step1, shown in the previous listing on lines 29 – 35 is responsible for calling the evenHandler's pre-step if defined, lines 32 – 34. This check is performed before each main step in the assembly process. We'll ignore it moving forward due to its redundancy. On line 35 we call the class'

`LoadAndParseJsonObjData` method which is responsible for parsing and loading all the data files associated with the ARM Thumb-1 instruction set.

Step 2, lines 37 – 43, logs the current step and then calls the `LinkJsonObjData` method on line 43. This method is responsible for linking the class' sub-objects to the actual associated Java class loaded from previous JSON files. Step 3 is handled on lines 45 – 54. Aside from the logging and event handler callbacks the code is very simple. The lexer is called on line 51 converting the source code into `ArtifactLines`. Notice that the first output file is written on line 53 if the `quellFileOutput` flag is set to false.

Pay special attention to the output files written during the assembly process. They give great insight into what the assembler is doing at the associated step in the process. Step 4, lines 56 – 65, is responsible for tokenizing the lexer output. This is done on line 62 with a call to the `TokenizeLexerArtifacts` method. Step 5, lines 67 – 76, brings us to our first line validation step. In this case the call to the `ValidateTokenizedLines` method, line 73, quickly checks to make sure the tokenizer results are all valid lines.

The second output file is written on line 75, if file output is not suppressed. This file contains validated, tokenized data and is a good place to validate your lexer and tokenizer steps. At this point in the assembly process, we have clean, tokenized data but there are a few things we need to adjust. Let's take a look at the code in the next listing.

#### *[Listing 18-4. AssemblerThumb.java Main Method Details 3](#)*

```
078 Logger.wrl("");
079 lastStep = 6;
080 Logger.wrl("STEP 6: Combine comment tokens as children of the initial
comment token");
081 if (eventHandler != null) {
082     eventHandler.RunAssemblerPreStep(lastStep, this);
083 }
084 CollapseCommentTokens(lastStep);
085
086 Logger.wrl("");
087 lastStep = 7;
```

```

088 Logger.wrl("STEP 7: Expand register ranges into individual register
entries");
089 if (eventHandler != null) {
090     eventHandler.RunAssemblerPreStep(lastStep, this);
091 }
092 ExpandRegisterRangeTokens(lastStep);
093
094 Logger.wrl("");
095 lastStep = 8;
096 Logger.wrl("STEP 8: Combine list and group tokens as children of the
initial list or group token");
097 if (eventHandler != null) {
098     eventHandler.RunAssemblerPreStep(lastStep, this);
099 }
100 CollapseListAndGroupTokens(lastStep);
101
102 Logger.wrl("");
103 lastStep = 9;
104 Logger.wrl("STEP 9: Mark OpCode, OpCode argument, and register
tokens");
105 if (eventHandler != null) {
106     eventHandler.RunAssemblerPreStep(lastStep, this);
107 }
108 PopulateOpCodeAndArgData(lastStep);
109
110 Logger.wrl("");
111 lastStep = 10;
112 Logger.wrl("STEP 10: Mark directive and directive argument tokens,
create area based line lists with hex numbering");
113 if (eventHandler != null) {
114     eventHandler.RunAssemblerPreStep(lastStep, this);
115 }
116 PopulateDirectiveArgAndAreaData(lastStep);
117 if (!quellFileOutput) {
118     Utils.WriteObject(asmDataTokened, "Assembly Tokenized Data",
"output_tokened_phase2_refactored.json", rootOutputDir);
119 }
120 ...

```

The AssemblerThumb class' RunAssembler method, part 3.

In the previously shown listing, part 3 of the `RunAssembler` method review, we'll take a look at steps 6 to 10 which take care of refactoring the structure of some `TokenLine` objects. On lines 78 – 84 we have step 6, where we handle grouping comment tokens with a call to the `CollapseCommentTokens` method on line 84. This call will adjust the structure of comment tokens for a given token line by moving all tokens subsequent from the initial comment token into sub-tokens by associating them with the initial comment token's payload.

The next step, step 7, takes care of another restructuring we need to perform. The code on lines 86 – 92 is responsible for the logging and event handler checks we've seen before. The call to the `ExpandRegisterRangeTokens` method converts register range opcode arguments into a series of register arguments. In essence we're flattening out the register ranges so that we can compare instruction arguments more easily.

Step 8, lines 94 – 100, is the last step where we perform a restructuring of the tokens that represent a line of assembly source code. In this case on line 100 we call the `CollapseListAndGroupTokens` method. This method is similar to the `CollapseCommentTokens` method call we saw a minute ago. This method alters the structure of group and list arguments such that the tokens that make up the list or group are now sub-tokens, aka sub-arguments, of the first group or list token.

We're doing the same thing we just did to comment tokens except with group and list tokens. The remaining two steps in this block of code are responsible for properly marking opcode, directive, and argument tokens. Step 9, lines 102 – 108, is responsible for calling the `PopulateOpCodeAndArgData` method which takes care of marking opcode and opcode argument tokens.

The next step, step 10, can be found on lines 110 – 119. A similar process to that of step 9 is performed with a call to the `PopulateDirectiveArgAndAreaData` method. This method is responsible for marking which tokens are directives or directive arguments. At this point in the method, we're done refactoring the data. A new file is written on line 118, the `output_tokened_phase2_refactored.json` file. This file expresses the final structure of the original assembly source code.

*Listing 18-5. AssemblerThumb.java Main Method Details 4*

```
121 Logger.wrl("");
122 lastStep = 11;
123 Logger.wrl("STEP 11: Validate OpCode lines against known OpCodes by
comparing arguments");
124 if (eventHandler != null) {
125     eventHandler.RunAssemblerPreStep(lastStep, this);
126 }
127 ValidateOpCodeLines(lastStep);
128
129 Logger.wrl("");
130 lastStep = 12;
131 Logger.wrl("STEP 12: Validate directive lines against known directives
by comparing arguments");
132 if (eventHandler != null) {
133     eventHandler.RunAssemblerPreStep(lastStep, this);
134 }
135 ValidateDirectiveLines(lastStep);
136 if (!quellFileOutput) {
137     Utils.writeObject(asmDataTokened, "Assembly Tokenized Data",
"output_tokened_phase3_valid_lines.json", rootOutputDir);
138     Utils.writeObject(symbols, "Symbol Data", "output_symbols.json",
rootOutputDir);
139 }
140
141 Logger.wrl("");
142 Logger.wrl("STEP 13: List Assembly Source Areas and Build Binary
Output:");
143 if (areaThumbCode != null) {
144     Logger.wrl("AreaThumbCode: Title: " + areaThumbCode.title);
145     Logger.wrl("AreaThumbCode: AreaLine: " + areaThumbCode.lineNumArea
+ " EntryLine: " + areaThumbCode.lineNumEntry + " EndLine: " +
areaThumbCode.lineNumEnd);
146     Logger.wrl("AreaThumbCode: Attributes: IsCode: " +
areaThumbCode.isCode + " IsData: " + areaThumbCode.isData + " IsReadOnly:
" + areaThumbCode.isReadOnly + " IsReadWrite: " +
areaThumbCode.isReadWrite);
147     if (!quellFileOutput) {
148         Utils.writeObject(asmAreaLinesCode, "Assembly Source Area Code
Lines", "output_area_lines_code.json", rootOutputDir);
149         Utils.writeObject(areaThumbCode, "Assembly Source Area Code
Desc", "output_area_desc_code.json", rootOutputDir);
```

```

150     }
151     BuildBinLines(lastStep, asmAreaLinesCode, areaThumbCode);
152 } else {
153     Logger.wrl("AreaThumbCode: is null");
154 }
155
156 if (areaThumbData != null) {
157     Logger.wrl("AreaThumbData: Title: " + areaThumbData.title);
158     Logger.wrl("AreaThumbData: AreaLine: " + areaThumbData.lineNumArea
+ " EntryLine: " + areaThumbData.lineNumEntry + " EndLine: " +
areaThumbData.lineNumEnd);
159     Logger.wrl("AreaThumbData: Attributes: IsCode: " +
areaThumbData.isCode + " IsData: " + areaThumbData.isData + " IsReadOnly:
" + areaThumbData.isReadOnly + " IsReadWrite: " +
areaThumbData.isReadWrite);
160     if (!quellFileOutput) {
161         Utils.writeObject(asmAreaLinesData, "Assembly Source Area Data
Lines", "output_area_lines_data.json", rootOutputDir);
162         Utils.writeObject(areaThumbData, "Assembly Source Area Data
Desc", "output_area_desc_data.json", rootOutputDir);
163     }
164     BuildBinLines(lastStep, asmAreaLinesData, areaThumbData);
165 } else {
166     Logger.wrl("AreaThumbData: is null");
167 }
168
169 if (!quellFileOutput) {
170     Utils.writeObject(asmDataTokened, "Assembly Tokenized Data",
"output_tokened_phase4_bin_output.json", rootOutputDir);
171 }
172
173 Logger.wrl("");
174 Logger.wrl("Assembler Meta Data:");
175 Logger.wrl("Title: " + assemblyTitle);
176 Logger.wrl("SubTitle: " + assemblySubTitle);
177 Logger.wrl("LineLengthBytes: " + lineLenBytes);
178 Logger.wrl("LineLengthWords: " + lineLenWords);
179 Logger.wrl("LineLengthHalfWords: " + lineLenHalfWords);
180 Logger.wrl("LineBitSeries:");
181 lineBitSeries.Print("\t");
182
183 if (eventHandler != null) {
184     eventHandler.RunAssemblerPost(this);

```

## The AssemblerThumb class' RunAssembler method, part 4.

The next block of code in this method, listed previously, handles steps 11 – 13. Step 11, lines 121 – 127, performs a line-by-line validation of opcodes and their arguments with a call to the `ValidateOpCodeLines` method on line 127. Step 12, lines 129 – 139, is similar to step 11 except that in this case we're validating directives and their arguments as opposed to opcodes. The method call, `ValidateDirectiveLines`, on line 135, is responsible for making sure any directives and their associated arguments are valid.

There are two more output files generated during step 12, if the `quellFileOutput` field is not false. On lines 137 and 138 the third version of the assembler's state is written followed by an output file containing the symbols found in the assembly source code. Step 13 has two parts to it. The first part runs from line 141 – 154 and is responsible for logging the details of the assembly source code' `@CODE` area. The lines and data associated with this area are written to files to help with the debugging process, lines 148 to 149.

The second part to step 13 runs from line 156 – 167. This block of code runs when there is a `@DATA` area defined in the assembly source code. The same information about the area is logged and two output files are generated. These files are similar to the output generated for the `@CODE` area except that they contain lines from the `@DATA` area, if defined. The calls to the `BuildBinLines` method on lines 151 and 164 prepare all the `TokenLines` in this `@CODE` and/or `@DATA` area for binary encoding

Lastly, on lines 169 – 171, the final version of the assembler's data is written to the `output_tokened_phase4_bin_output.json` file. The remaining lines of code, 173 – 181, logs some useful information about the assembly source code's instruction set and on line 184 we call the `RunAssemblerPost` method to allow for any customized code to run after the assembly process completes.

### *Listing 18-6. AssemblerThumb.java Main Method Details 5*

```
186     } catch (Exception e) {
187         Logger.wrlErr("Error in RunAssembler method on step: " +
lastStep);
188         if (lastLine != null) {
189             Logger.wrlErr("Last line processed: " + lastLine.lineNumAbs);
190             if (lastLine.source != null) {
191                 Logger.wrlErr("Last line source: " +
lastLine.source.source);
192             }
193         } else {
194             Logger.wrlErr("Last line processed: unknown");
195             Logger.wrlErr("Last line source: unknown");
196         }
197
198         if (lastToken != null) {
199             Logger.wrlErr("Last token processed: " + lastToken.source + " with index " + lastToken.index + ", type name '" + lastToken.type_name + "'", and line number " + lastToken.lineNumAbs);
200         } else {
201             Logger.wrlErr("Last token processed: unknown");
202         }
203         throw e;
204     }
205 }
```

The AssemblerThumb class' RunAssembler method, part 5.

The remaining code in this method is meant to process any exceptions encountered during the assembly process. There's not a lot to discuss here but there are some deceptively simple lines of code in this snippet. Does anything stand out to you? One thing you might have noticed is that the exception handler relies on a few class fields, lastStep, lastLine, and lastToken to report the error.

As I mentioned earlier, and as we'll soon see during the next few sections, the main methods in this class register the work they are doing by updating the afore mentioned class fields on each new loop iteration during their respective work. This implementation approach lets us report information

about where an error occurred from any main method in a centralized way. This comes in handy when dealing with any errors that pop-up during development. That brings us to the end of this method review.

Using it as a template for the remaining main methods will give us a path of exploration through the core of the assembly process. As such I'll try to refer back to this method and the associated step when reviewing the class' remaining main methods. The next main method we'll look at is the LoadAndParseJsonObjData method step 1. It's responsible for loading up all the JSON files specified as belonging to this instruction set.

## Main Method Details: AssemblerThumb.java - Step 1

The first step in the assembly process, as outlined by the main execution method, is to load and parse associated JSON data files. The main method responsible for handling this, as you may have guessed, is the LoadAndParseJsonObjData method. We've seen bits and pieces of this method in demonstration sections prior to this chapter. Let's see what we've got!

*Listing 18-7. AssemblerThumb.java Main Method Details 6*

```
001 public void LoadAndParseJsonObjData(int step) throws
ExceptionNoEntryFound, ExceptionLoader, IOException,
ClassNotFoundException, InstantiationException, IllegalAccessException,
NoSuchMethodException, InvocationTargetException {
002     if (eventHandler != null) {
003         eventHandler.LoadAndParseJsonObjDataPre(step, this);
004     }
005
006     Logger.wrl("AssemblerThumb: LoadAndParseJsonObjData");
007     Class cTmp = null;
008     Loader ldr = null;
009     String json = null;
010     String jsonName = null;
011     JsonObj jsonObj = null;
012
013     for (JsonObjIsFile entry : isaDataSet.is_files) {
014         if (eventHandler != null) {
```

```

015         eventHandler.LoadAndParseJsonObjDataLoopPre(step, this,
entry);
016     }
017
018     cTmp = Class.forName(entry.loader_class);
019     ldr = (Loader) cTmp.getDeclaredConstructor().newInstance();
020     json = null;
021     jsonName = null;
022     jsonObj = null;
023
024     isaLoader.put(entry.loader_class, ldr);
025     Logger.wrl("AssemblerThumb: RunAssembler: Loader created '" +
entry.loader_class + "'");
026
027     json = FileLoader.LoadStr(entry.path);
028     jsonSource.put(entry.path, json);
029     Logger.wrl("AssemblerThumb: RunAssembler: Json loaded '" +
entry.path + "'");
030
031     jsonObj = ldr.ParseJson(json, entry.target_class, entry.path);
032     Logger.wrl("AssemblerThumb: RunAssembler: " +
jsonObj.GetName());
033     jsonName = jsonObj.GetName();
034     isaData.put(jsonName, jsonObj);
035     Logger.wrl("AssemblerThumb: RunAssembler: Json parsed as '" +
entry.target_class + "'");
036     Logger.wrl("AssemblerThumb: RunAssembler: Loading isaData with
entry '" + jsonName + "'");
037
038     if
(jsonObj.GetLoader().equals("net.middlemind.GenAsm.Loaders.Thumb.LoaderIsE
ntryTypes")) {
039         jsonObjIsEntryTypes = (JsonObjIsEntryTypes) jsonObj;
040         Logger.wrl("AssemblerThumb: RunAssembler: Found
JsonObjIsEntryTypes object, storing it...");
041
042     } else if
(jsonObj.GetLoader().equals("net.middlemind.GenAsm.Loaders.Thumb.LoaderIsValidLines")) {
043         jsonObjIsValidLines = (JsonObjIsValidLines) jsonObj;
044         Logger.wrl("AssemblerThumb: RunAssembler: Found
JsonObjIsValidLines object, storing it...");
045

```

```

046      } else if
(jsonObj.GetLoader().equals("net.middlemind.GenAsm.Loaders.Thumb.LoaderIsEmptyDataLines")) {
047          jsonObjIsEmptyDataLines = (JsonObjIsEmptyDataLines) jsonObj;
048          Logger.wrl("AssemblerThumb: RunAssembler: Found
JsonObjIsEmptyDataLines object, storing it..."); 
049
050      } else if
(jsonObj.GetLoader().equals("net.middlemind.GenAsm.Loaders.Thumb.LoaderIsOpCodes")) {
051          jsonObjIsOpCodes = (JsonObjIsOpCodes) jsonObj;
052          lineBitSeries = jsonObjIsOpCodes.bit_series;
053
054          lineNumRange = new JsonObjNumRange();
055          lineNumRange.alignment = "WORD";
056          lineNumRange.bcd_encoding = false;
057          lineNumRange.bit_len = lineBitSeries.bit_len;
058          lineNumRange.obj_name = "JsonObjNumRange";
059          lineNumRange.min_value = 0;
060          lineNumRange.max_value = 65536;
061          lineNumRange.ones_complement = false;
062          lineNumRange.twos_complement = false;
063
064          pcPrefetchBytes = jsonObjIsOpCodes.pc_prefetch_bytes;
065          pcPrefetchHalfwords = jsonObjIsOpCodes.pc_prefetch_halfwords;
066          pcPrefetchWords = jsonObjIsOpCodes.pc_prefetch_words;
067
068          if
(jsonObjIsOpCodes.endian.equals(AssemblerThumb.ENDIAN_NAME_BIG)) {
069              isEndianBig = true;
070              isEndianLittle = false;
071          } else {
072              isEndianBig = false;
073              isEndianLittle = true;
074          }
075          Logger.wrl("AssemblerThumb: RunAssembler: Found
JsonObjIsOpCodes object, storing it..."); 
076
077      } else if
(jsonObj.GetLoader().equals("net.middlemind.GenAsm.Loaders.Thumb.LoaderIsDirectives")) {
078          jsonObjIsDirectives = (JsonObjIsDirectives) jsonObj;

```

```

079         Logger.wrl("AssemblerThumb: RunAssembler: Found
JsonObjIsDirectives object, storing it...");
080
081     } else if
(jsonObj.GetLoader().equals("net.middlemind.GenAsm.Loaders.Thumb.LoaderIsRegisters")) {
082         jsonObjIsRegisters = (JsonObjIsRegisters) jsonObj;
083         Logger.wrl("AssemblerThumb: RunAssembler: Found
JsonObjIsRegisters object, storing it...");
084     }
085
086     if (eventHandler != null) {
087         eventHandler.LoadAndParseJsonObjDataLoopPost(step, this,
entry);
088     }
089 }
090
091     if (jsonObjIsEntryTypes == null) {
092         throw new ExceptionNoEntryFound("Could not find required
JsonObjIsEntryTypes instance.");
093
094     } else if (jsonObjIsValidLines == null) {
095         throw new ExceptionNoEntryFound("Could not find required
JsonObjIsValidLines instance.");
096
097     } else if (jsonObjIsOpCodes == null) {
098         throw new ExceptionNoEntryFound("Could not find required
JsonObjIsOpCodes instance.");
099     }
100
101     if (eventHandler != null) {
102         eventHandler.LoadAndParseJsonObjDataPost(step, this);
103     }
104 }
```

The AssemblerThumb class' LoadAndParseJsonObjData method. This method is responsible for loading all instruction set data files.

The method starts off on lines 2 – 4 with a call to the event handler to allow for customizations before the method's work begins if the

`eventHandler` field is defined. Local variables are initialized on lines 7 – 11. These are temporary variables that provide support for loading and parsing JSON files. In this case we’re going to be working through all the files listed in the instruction set’s definition. We iterate through the list of files on line 13 with further support for customization provided on lines 14 – 16.

On line 18 the `cTmp` variable holds the full name of the specified loader class to use for this file. The loader is then initialized on line 19 and the loop variables `json`, `jsonName`, and `jsonObj` are all reset for this loop iteration, lines 20 – 22. As we’ve seen during our review of the class fields, we keep a decent amount of information around regarding the data we’re loading.

Next up, on line 24, we register the file loader in the `isLoader` field by the full name of the class. The contents of the file are loaded and stored in the `json` local variable on line 27, and the data is stored in the `jsonSource` field on line 28. The text-based data is then converted into a Java object on line 31 and the resulting data is stored in the `isaData` field, by name, on line 34. We register all data files loaded in the `isaData` field so we can reference them by name when needed. The block of code on lines 38 to 84 is responsible for keeping direct references to important data, discerned by the name of the loader class used to parse the data file.

Note that the opcodes section, lines 50 – 75, prepares a few class fields for use during the assembly process. Information about the numeric range allowed for a given line of assembly as well as the prefetch is defined on lines 54 – 66. The default endianness of the assembler is set on lines 68 – 74. Customization is supported via the event handler callback on lines 87 and 102. There are a few sanity checks performed on lines 91 – 99 to ensure key data is present. Now that we have the Thumb-1 instruction set data files loaded, we need to do some linking.

## Main Method Details: AssemblerThumb.java - Step 2

Because the data is loaded from JSON files, connections between JSON objects are created with string references to object names. Once the data is loaded into holder classes we want to scan through the files and make sure to connect any key references by calling the class’ `Link` method.

*Listing 18-8. AssemblerThumb.java Main Method Details 7*

```

01 public void LinkJsonObjData(int step) throws ExceptionJsonObjLink {
02     if (eventHandler != null) {
03         eventHandler.LinkJsonObjDataPre(step, this);
04     }
05
06     Logger.wrl("AssemblerThumb: LinkJsonObjData");
07     for (String s : isaData.keySet()) {
08         if (eventHandler != null) {
09             eventHandler.LinkJsonObjDataLoopPre(step, this, s);
10         }
11
12         JsonObject jsonObj = isaData.get(s);
13         jsonObj.Link(jsonObjIsEntryTypes);
14
15         if (eventHandler != null) {
16             eventHandler.LinkJsonObjDataLoopPost(step, this, s);
17         }
18     }
19
20     if (eventHandler != null) {
21         eventHandler.LinkJsonObjDataPost(step, this);
22     }
23 }
```

The AssemblerThumb class' `LinkJsonObjData` method. This method is responsible for connecting holder classes by name.

Note that the method starts off with an event handler callback check, lines 2 – 4. We'll pay less attention to these event handler callbacks due to their simplicity and redundancy. The loop on lines 7 – 18 iterates over the contents of the `isaData` map that we populated during the data load. A local variable is initialized based on the stored reference, line 12, and the holder class' `Link` method is called on line 13. In this way each holder class associated with the instruction set will be properly linked.

## Main Method Details: AssemblerThumb.java - Step 3 and 4

Steps three and four of the assembly process see the loaded source code processed by the lexer and tokenizer. We've seen this code in action before, during our review of the lexer and tokenizer classes. We've also seen specifically how the `AssemblerThumb` class uses the `LexerizeAssemblySource` and `TokenizeLexerArtifacts` methods to process the original assembly source code during our review of the class' support methods. We won't review this code again here, rather we'll move on to the next section and take a look at the operative methods of assembly step 5.

## Main Method Details: AssemblerThumb.java - Step 5

In step 5 of the assembly process, we perform our first, simple, scan to validate the lines that we've loaded up. This is the last step before the refactoring work that's done in steps 6 to 12. The pertinent method call made in this step is the call to the `ValidateTokenizedLines` method. Let's take a look at this method now.

*Listing 18-9. AssemblerThumb.java Main Method Details 8*

```
01 public boolean ValidateTokenizedLines(int step) throws
ExceptionNoValidLineFound {
02     if (eventHandler != null) {
03         eventHandler.ValidateTokenizedLinesPre(step, this);
04     }
05
06     Logger.wrl("AssemblerThumb: ValidateTokenizedLines");
07     for (TokenLine line : asmDataTokened) {
08         lastLine = line;
09         if (eventHandler != null) {
10             eventHandler.ValidateTokenizedLinesLoopPre(step, this, line);
11         }
12
13         if (!ValidateTokenizedLine(step, line, jsonObjIsValidLines,
jsonObjIsValidLines.is_valid_lines.get(JsonObjIsValidLines.LINE_EMPTY))) {
```

```

14         throw new ExceptionNoValidLineFound("Could not find a matching
valid line for line number, " + line.lineNumAbs + " with source text, '" +
line.source.source + "'");
15     }
16
17     if (eventHandler != null) {
18         eventHandler.ValidateTokenizedLinesLoopPost(step, this, line);
19     }
20 }
21
22 if (eventHandler != null) {
23     eventHandler.ValidateTokenizedLinesPost(step, this);
24 }
25
26 return true;
27 }

```

```

01 public boolean ValidateTokenizedLine(int step, TokenLine line,
JsonObjIsValidLines validLines, jsonObjIsValidLine validLineEmpty) {
02     if (eventHandler != null) {
03         eventHandler.ValidateTokenizedLinePre(step, this, line,
validLines, validLineEmpty);
04     }
05
06     int tokenCount = line.payload.size();
07     int[] res = null;
08     int currentEntry = -1;
09     int entries = -1;
10
11     if (tokenCount == 0) {
12         line.validLineEntry = validLineEmpty;
13         line.isEmpty = true;
14         return true;
15     }
16
17     for ( jsonObjIsValidLine validLine : validLines.is_valid_lines) {
18         res = null;
19         currentEntry = -1;
20         tokenCount = line.payload.size();
21         entries = -1;
22
23         for (Token token : line.payload) {
24             lastToken = token;

```

```

25
26         res = FindValidLineEntryWithMatchingTokenType(validLine,
27 token, currentEntry, 0);
28         if (res == null) {
29             break;
30         }
31         if (currentEntry == -1) {
32             entries = 1;
33             currentEntry = res[0];
34             tokenCount--;
35         } else {
36             if (res[0] > currentEntry) {
37                 entries++;
38             }
39             if (res[0] >= currentEntry) {
40                 currentEntry = res[0];
41                 tokenCount--;
42             } else {
43                 break;
44             }
45         }
46     }
47 }
48
49     if (tokenCount == 0 && entries == validLine.is_valid_line.size())
{
50         line.validLineEntry = validLine;
51         return true;
52     }
53 }
54
55     if (eventHandler != null) {
56         eventHandler.ValidateTokenizedLinePost(step, this);
57     }
58
59     return false;
60 }
```

A listing of the pertinent methods called during step 5 of the assembly process.

The initial line validation process, performed during this step, requires two methods to scan the current list of `TokenLine` objects and find any unknown line configurations. Due to the simplicity of the assembly source code syntax, we can determine all the different valid lines that can exist and even list the entry types allowed for `Tokens` on said line. The first method in the set, `ValidateTokenizedLines`, is used to scan all the tokenized source code lines. From this point on we're going to be mainly working with `TokenLines` and `Tokens` as they represent the highest-level object that can be tied directly back to a string of assembly source code.

On line 7 we iterate over the list of tokenized lines and for each line we call the `ValidateTokenizedLine` method to perform the necessary checks. The method receives a few arguments. The current step, `step`, the current line, `line`, a list of valid lines, `validLines`, and a special empty line object, `validEmptyLine`. The method variables are prepared on lines 6 – 9. The if statement on lines 11 – 15 is there to detect an empty line. If no tokens are found on the current line, then the argument, `validLineEmpty`, is used as the matching valid line entry and the `isLineEmpty` flag is set to true.

If tokens are present, we iterate over the list of valid lines, line 17, and set loop variables on lines 18 – 21. Note that we set the `tokenCount` to the number of tokens available. We'll use this variable to count down to zero, indicating an argument for argument match. For each token line we scan the contents of the payload starting on line 23. The support method, `FindValidLineEntryWithMatchingTokenEntryType` – line 26 – supports successive calls by passing in the `currentEntry` value. This value serves to set the starting position for checking an argument match.

On line 27 – 29, we exit the loop if the support method returns a null value for a match. Next up, on lines 31 – 35, for the first loop iteration only, we set the current entry and decrement the `tokenCount` variable. Recall from our review of the support methods that the returned value is an array of integers with the first integer indicating the index of the match. We check to make sure the match happened with the next argument, line 36 – 38, not the one we're currently on. If so, the `entries` variable is incremented.

For the case where we are scanning along valid entries after the first, lines 40 – 43, we update the `currentEntry` variable and decrement the `tokenCount`. If the match is found to be from before the current position, we

exit the inner for loop, line 44. The method exit condition is on lines 49 – 52. If we have processed all the tokens and we've scanned through all the entries in the current valid line, then we have a match and we can return from the method on line 51.

## Main Method Details: AssemblerThumb.java - Step 6

Step 6 of the assembly process brings us to our first refactoring method. The operative method called during this step in the process is the `CollapseCommentTokens` method. The purpose of this method is to “collapse” comment tokens. To us this means that all comment tokens, subsequent to the first comment token, are made sub-tokens or children of the first comment token. Let's take a look at how it works.

*Listing 18-10. AssemblerThumb.java Main Method Details 9*

```
01 public void CollapseCommentTokens(int step) {  
02     if (eventHandler != null) {  
03         eventHandler.CollapseCommentTokensPre(step, this);  
04     }  
05  
06     Logger.wrl("AssemblerThumb: CollapseCommentTokens");  
07     boolean inComment = false;  
08     Token commentRoot = null;  
09     List<Token> clearTokens = null;  
10  
11     for (TokenLine line : asmDataTokened) {  
12         lastLine = line;  
13         if (eventHandler != null) {  
14             eventHandler.CollapseCommentTokensLoopPre(step, this, line);  
15         }  
16  
17         inComment = false;  
18         commentRoot = null;  
19         clearTokens = new ArrayList<>();  
20  
21         for (Token token : line.payload) {  
22             lastToken = token;  
23             if (token.type_name.equals(JsonObjIsEntryTypes.NAME_COMMENT))  
{  
24                 if (!inComment) {
```

```

25         commentRoot = token;
26         commentRoot.payload = new ArrayList<>();
27         inComment = true;
28     } else {
29         token.index = commentRoot.payload.size();
30         commentRoot.payload.add(token);
31         clearTokens.add(token);
32     }
33 }
34 }
35
36 if (inComment) {
37     commentRoot.payloadLen = commentRoot.payload.size();
38     line.payload.removeAll(clearTokens);
39     line.payloadLen = line.payload.size();
40 }
41
42 if (eventHandler != null) {
43     eventHandler.CollapseCommentTokensLoopPost(step, this, line);
44 }
45 }
46
47 if (eventHandler != null) {
48     eventHandler.CollapseCommentTokensPost(step, this);
49 }
50 }

```

The CollapseCommentTokens method from the AssemblerThumb class.

Method variables are initialized on lines 7 – 9. The type of scan done by this method iterates over all tokens for each `TokenLine` tracking if we've encountered a comment token. Comments in assembly are not quite like comments in higher level programming languages. Assembly comments only work on one line and every character after the comment is considered part of the comment.

Iterating over all the token lines, line 11, we reset the method variables for the work we have to do on the current line. The `inComment` flag is set to false, the `commentRoot` token is set to null, and the list of tokens to clear is reset, lines 17 – 19. Next up, on line 21 we iterate over all the tokens on the

current line looking for a comment token. Notice that the `lastLine` and `lastToken` class fields are updated by this method on lines 12 and 22 respectively.

A comment token is detected via the comparison on line 23. Recall from our review of the `JsonObjIsEntryTypes` class that there are a number of static class fields to help identify tokens. On lines 24 – 32 we handle processing the comment tokens. If we’re not currently in a comment, we set the `commentRoot`, we reset the `payload`, and we set the `inComment` flag to true, lines 25 – 27. If we are in a comment, we set the `index` field of the token based on the size of the comment root token’s `payload`.

The new comment token is added to the `commentRoot` token’s `payload` and the `clearTokens` method variable. At the end of the line’s token scan if a comment was found, lines 36 – 40, we update the `commentRoot` token’s `payloadLen` field and then we remove the tokens from the `tokenLine` object on line 38. Lastly, we update the `tokenLine`’s `payloadLen` field to reflect the shift of tokens. Note that there are two callback methods that support customizing the method on lines 43 and 48.

## Main Method Details: AssemblerThumb.java - Step 7

In step 7 of the assembly process, we undertake another refactoring of the data structure that represents our assembly source code. This method is a little lengthy, so I’ll break it up into three blocks of code. Take a moment to note the way we handle these token scans as you read through the code.

*Listing 18-11. AssemblerThumb.java Main Method Details 10*

```
001 public void ExpandRegisterRangeTokens(int step) throws
ExceptionNoEntryFound, ExceptionMalformedRange {
002     if (eventHandler != null) {
003         eventHandler.ExpandRegisterRangeTokensPre(step, this);
004     }
005
006     Logger.wrl("AssemblerThumb: ExpandRegisterRangeToken");
007     int rangeRootIdxLow = -1;
008     int rangeRootIdxHi = -1;
009     Token rangeRootLow = null;
```

```

010     Token rangeRootHi = null;
011     String rangeStr = null;
012     int[] range = null;
013     int count = -1;
014     Token newToken = null;
015     int i = -1;
016
017     List<Token> rangeAddTokensLow = null;
018     List<Token> rangeAddTokensHi = null;
019     JsonObjectIsEntryType entryTypeRegLow = null;
020     JsonObjectIsEntryType entryTypeRegHi = null;
021
022     for (TokenLine line : asmDataTokened) {
023         lastLine = line;
024         if (eventHandler != null) {
025             eventHandler.ExpandRegisterRangeTokensLoopPre(step, this,
026             line);
027         }
028         rangeRootIdxLow = 0;
029         rangeRootIdxHi = 0;
030         rangeRootLow = null;
031         rangeRootHi = null;
032         rangeStr = "";
033         range = null;
034         count = 0;
035         newToken = null;
036         i = 0;
037
038         rangeAddTokensLow = new ArrayList<>();
039         rangeAddTokensHi = new ArrayList<>();
040         entryTypeRegLow =
FindEntryType(JsonObjIsEntryTypes.NAME_REGISTER_LOW);
041         entryTypeRegHi =
FindEntryType(JsonObjIsEntryTypes.NAME_REGISTER_HI);
042
043         for (Token token : line.payload) {
044             lastToken = token;
045             CleanTokenSource(token);
046             ...

```

Part 1 of the AssemblerThumb class' ExpandRegisterRangeTokens method listing.

The first part of the `ExpandRegisterRangeTokens` method, shown in the previous listing is responsible for preparing the method variables and starting the scan of the token lines. On lines 7 – 15 the method variables are initialized. Let's go over their meaning. The first two variables are used to track the index of the root of a register range set of tokens. Subsequently, the `rangeRootLow` and `rangeRootHi` variables are used to store a reference to the start of a low and high range expression, respectively.

The remaining five variables are temporary variables used by the method. The next set of method variable initializations, lines 17 – 20, are used to store and identify register range tokens. A register range is a compound expression that combines what would normally be a series of tokens into just one token. Ranges have a starting register, a range delimiter, and an ending register. An example in assembly source code is as follows.

```
Before: PUSH{R0-R4, LR}  
After:  PUSH{R0,R1,R2,R3,R4, LR}
```

As you can see from the example, we're going to be adding more tokens into the token line that holds the register range. To hold this new data the list variables on lines 17 – 18 are declared, followed by two variables that are used to find register range tokens, the `entryTypeRegLow` and `entryTypeRegHi` variables on lines 19 and 20. We iterate over the token lines on line 22. Notice that we update the `lastLine` class field to make sure we know what line we're looking at if an unhandled exception is encountered.

On lines 28 – 36 the method variables are reset for the current `TokenLine` object and on lines 38 – 41 we prepare the token lists and entry type variables used to store new tokens and find register hi or low tokens respectively. Next up on line 43 we iterate over the tokens for the current token line. Note that we also update the class' `lastToken` field so that we know what token we're working on in the case of an exception, line 44. The token's assembly source code is cleaned on line 45 with a call to the support method we reviewed earlier in the text. We'll pick things up with the contents of the inner for loop in the subsequent listing.

*Listing 18-12. AssemblerThumb.java Main Method Details 11*

```
047 if
(token.type_name.equals(JsonObjIsEntryTypes.NAME_REGISTER_RANGE_LOW)) {
048     rangeRootLow = token;
049     rangeRootIdxLow = rangeRootLow.index;
050     rangeStr = rangeRootLow.source;
051     rangeStr = CleanRegisterRangeString(rangeStr,
JsonObjIsRegisters.CHAR_RANGE);
052     range = Utils.GetIntsFromRange(rangeStr,
JsonObjIsRegisters.CHAR_RANGE);
053     count = 0;
054
055     for (i = range[0]; i <= range[1]; i++) {
056         newToken = new Token();
057         newToken.artifact = rangeRootLow.artifact;
058         newToken.index = rangeRootLow.index + count;
059         newToken.lineNumAbs = rangeRootLow.lineNumAbs;
060         newToken.payload = new ArrayList<>();
061         newToken.payloadLen = 0;
062         newToken.source = JsonObjIsRegisters.CHAR_START + i;
063         newToken.type = entryTypeRegLow;
064         newToken.type_name = entryTypeRegLow.type_name;
065         newToken.isOpCodeArg = true;
066         rangeAddTokensLow.add(newToken);
067         count++;
068     }
069
070 } else if
(token.type_name.equals(JsonObjIsEntryTypes.NAME_REGISTER_RANGE_HI)) {
071     rangeRootHi = token;
072     rangeRootIdxHi = rangeRootHi.index;
073     rangeStr = rangeRootHi.source;
074     rangeStr = CleanRegisterRangeString(rangeStr,
JsonObjIsRegisters.CHAR_RANGE);
075     range = Utils.GetIntsFromRange(rangeStr,
JsonObjIsRegisters.CHAR_RANGE);
076     count = 0;
077
078     for (i = range[0]; i <= range[1]; i++) {
079         newToken = new Token();
080         newToken.artifact = rangeRootHi.artifact;
081         newToken.index = rangeRootHi.index + count;
```

```

082     newToken.lineNumAbs = rangeRootHi.lineNumAbs;
083     newToken.payload = new ArrayList<>();
084     newToken.payloadLen = 0;
085     newToken.source = JsonObjIsRegisters.CHAR_START + i;
086     newToken.type = entryTypeRegHi;
087     newToken.type_name = entryTypeRegHi.type_name;
088     newToken.isOpCodeArg = true;
089     rangeAddTokensHi.add(newToken);
090     count++;
091 }
092 } ...

```

Part 2 of the AssemblerThumb class' ExpandRegisterRangeTokens method listing.

We'll pick things up where we left off on line 47 of the method's inner for loop. The previously listed code block has two main sections, one that handles the case of the low register range and one that handles the case of the hi register range. This block of code executes while we iterate over the line's tokens. On line 47 if we detect a token that has a low register range entry type, we prepare the loop variables on lines 48 – 53.

We keep track of the register range's root token, line 48, and the index of the token in the current line's payload, line 49. The method variable `rangeStr` is used on lines 50 – 51 to create a clean representation of the register range. Next, on line 52 the register range string is parsed, and two integers are returned, the starting and ending register indices. The `count` method variable is reset on line 53 as we prepare to loop over the register range indices and add new register tokens to represent each register in the range.

New register tokens are created for the missing low registers on lines 55 – 68. Notice that we do nothing with the new register tokens except save them in the `rangeAddTokensLow` list. We'll see how they are used in just a bit. The same process is performed on lines 70 – 92 for the register range hi tokens. Again, we keep track of the newly created register tokens by adding them to the `rangeAddTokensHi` list on line 89.

*Listing 18-13. AssemblerThumb.java Main Method Details 12*

```

095 if (rangeRootHi != null && rangeAddTokensHi != null &&
rangeAddTokensHi.size() > 0) {
096     line.payload.addAll(rangeRootIdxHi + 1, rangeAddTokensHi);
097     line.payload.remove(rangeRootHi);
098     line.payloadLen = line.payload.size();
099
100    count = 0;
101    for (Token token : line.payload) {
102        lastToken = token;
103        token.index = count;
104        count++;
105    }
106 }
107
108 if (rangeRootLow != null && rangeAddTokensLow != null &&
rangeAddTokensLow.size() > 0) {
109     line.payload.addAll(rangeRootIdxLow + 1, rangeAddTokensLow);
110     line.payload.remove(rangeRootLow);
111     line.payloadLen = line.payload.size();
112
113    count = 0;
114    for (Token token : line.payload) {
115        lastToken = token;
116        token.index = count;
117        count++;
118    }
119 } ...

```

Part 3 of the AssemblerThumb class' ExpandRegisterRangeTokens method listing.

The last block of code from this method's review is responsible for handling the newly created register tokens, if any, that are set to replace the register range token. We'll pick the review back up on line 95 of the method shown in the previous listing. On lines 95 – 106, if a high register range has been detected we have to inject the new register tokens back into the line's payload. This is handled on line 96. We insert the new register tokens behind the register range's starting token. Then on line 97 we remove the starting token and update the token line's payloadLen field.

We still have a little bit of work to do with the newly added tokens. Lines 100 – 105 make sure to re-index the line's payload of tokens so that the newly added tokens have correct `index` field values. A similar process is performed on lines 108 – 119 for register range `lo` tokens. This code is very similar to the code we've just reviewed and as such I'll leave it to you to review. Make sure you understand what's going on here before you move on. That brings us to the conclusion of assembly step 7's review. In the next review section, we'll take a look at the last refactoring step where we collapse group and list tokens in much the same way we did for the comment tokens.

## Main Method Details: AssemblerThumb.java - Step 8

Step 8 of the assembly process is the last step where we refactor the structure of the tokens. In this step the operative method, `CollapseListAndGroupTokens`, is called to reorganize list and group tokens in much the same way we adjusted the comment tokens.

*Listing 18-14. AssemblerThumb.java Main Method Details 13*

```
001 public void CollapseListAndGroupTokens(int step) throws
ExceptionNoClosingBracketFound, ExceptionListAndGroup {
002     if (eventHandler != null) {
003         eventHandler.CollapseListAndGroupTokensPre(step, this);
004     }
005
006     Logger.wrl("AssemblerThumb: CollapseListAndGroupTokens");
007     Token rootStartList = null;
008     int rootStartIdxList = -1;
009     Token rootStartGroup = null;
010     int rootStartIdxGroup = -1;
011
012     Token rootStopList = null;
013     int rootStopIdxList = -1;
014     Token rootStopGroup = null;
015     int rootStopIdxGroup = -1;
016
017     List<Token> clearTokensList = null;
018     List<Token> clearTokensGroup = null;
019     int copyStart = -1;
020     int copyEnd = -1;
021     int copyLen = -1;
```

```

022
023     for (TokenLine line : asmDataTokened) {
024         lastLine = line;
025         if (eventHandler != null) {
026             eventHandler.CollapseListAndGroupTokensLoopPre(step, this,
027             line);
028         }
029         rootStartList = null;
030         rootStartIdxList = -1;
031         rootStartGroup = null;
032         rootStartIdxGroup = -1;
033
034         rootStopList = null;
035         rootStopIdxList = -1;
036         rootStopGroup = null;
037         rootStopIdxGroup = -1;
038
039         clearTokensList = new ArrayList<>();
040         clearTokensGroup = new ArrayList<>();
041         copyStart = -1;
042         copyEnd = -1;
043         copyLen = -1;
044         ...

```

The first part of the AssemblerThumb class' CollapseListAndGroupTokens method.

Take a look at the start of the CollapseListAndGroupTokens method, shown in the previous listing. The method's variables are declared on lines 7 – 21. These variables are very similar to the method variables used by the expand register range process, and to a lesser extent the collapse comments process. On lines 7 – 10 we have variables that are used to track the index and root of the list and group start tokens and on lines 12 – 15 method variables are declared to track the index and root of the list and group stop tokens.

Subsequently, on line 17 – 21, method variables are declared to keep track of the tokens that must be cleared from the token line's payload. Remember, this refactoring is similar to what we did for the comment tokens. We are going to move group and list tokens, that follow the starting token,

from the token line's payload to the starting token's payload. This effectively makes them sub-tokens, sub-arguments, of the starting list or group token.

The loop to iterate over the tokenized assembly source code starts on line 23. Notice that we update the class' `lastLine` field, line 24, to keep track of what token line we're working with in the case of an unhandled exception. The method variables are reset for this loop iteration on lines 29 – 43. The next listing, shown subsequently, picks up where we left off on line 45 of the method.

*Listing 18-15. AssemblerThumb.java Main Method Details 14*

```
045 for (Token token : line.payload) {  
046     lastToken = token;  
047     if (token.type_name.equals(JsonObjIsEntryTypes.NAME_START_LIST)) {  
048         rootStartList = token;  
049         rootStartIdxList = rootStartList.index;  
050     } else if  
(token.type_name.equals(JsonObjIsEntryTypes.NAME_START_GROUP)) {  
052         rootStartGroup = token;  
053         rootStartIdxGroup = rootStartGroup.index;  
054     } else if  
(token.type_name.equals(JsonObjIsEntryTypes.NAME_STOP_LIST)) {  
056         rootStopList = token;  
057         rootStopIdxList = rootStopList.index;  
058     } else if  
(token.type_name.equals(JsonObjIsEntryTypes.NAME_STOP_GROUP)) {  
060         rootStopGroup = token;  
061         rootStopIdxGroup = rootStopGroup.index;  
062     }  
063 }  
064 }  
065  
066 if ((rootStartIdxList != -1 && rootStopIdxList == -1) ||  
(rootStartIdxList == -1 && rootStopIdxList != -1)) {  
067     throw new ExceptionNoClosingBracketFound("Could not find closing  
bracket for list.");  
068 }
```

```

069 } else if ((rootStartIdxGroup != -1 && rootStopIdxGroup == -1) ||
070   (rootStartIdxGroup == -1 && rootStopIdxGroup != -1)) {
071   throw new ExceptionNoClosingBracketFound("Could not find closing
bracket for group.");
072 } else if (rootStartIdxList != -1 && rootStopIdxList != -1 &&
rootStartIdxGroup != -1 && rootStopIdxGroup != -1) {
073   throw new ExceptionListAndGroup("Found list and group entries when
only one is allowed.");
074
075 } else if (rootStartIdxList != -1 && rootStopIdxList != -1) {
076   copyStart = (rootStartIdxList + 1);
077   copyEnd = rootStopIdxList;
078   copyLen = (copyEnd - copyStart);
079   for (int i = copyStart; i <= (copyStart + copyLen); i++) {
080     clearTokensList.add(line.payload.get(i));
081   }
082   line.payload.removeAll(clearTokensList);
083   line.payloadLen = line.payload.size();
084   rootStartList.payload.addAll(clearTokensList);
085   rootStartList.payloadLen = clearTokensList.size();
086
087   int count = 0;
088   for (Token token : rootStartList.payload) {
089     lastToken = token;
090     token.index = count;
091     count++;
092   }
093
094 } else if (rootStartIdxGroup != -1 && rootStopIdxGroup != -1) {
095   copyStart = (rootStartIdxGroup + 1);
096   copyEnd = rootStopIdxGroup;
097   copyLen = (copyEnd - copyStart);
098   for (int i = copyStart; i <= (copyStart + copyLen); i++) {
099     clearTokensGroup.add(line.payload.get(i));
100   }
101   line.payload.removeAll(clearTokensGroup);
102   line.payloadLen = line.payload.size();
103   rootStartGroup.payload.addAll(clearTokensGroup);
104   rootStartGroup.payloadLen = clearTokensGroup.size();
105
106   int count = 0;
107   for (Token token : rootStartGroup.payload) {

```

```
108     lastToken = token;  
109     token.index = count;  
110     count++;  
111 }  
112 }
```

The second part of the AssemblerThumb class' CollapseListAndGroupTokens method listing.

The block of code in the previous listing shows the method's inner for loop. This part of the method is responsible for scanning the tokens of a given line of assembly source code to look for list or group start tokens. The class field, `lastToken`, is updated on line 46 to keep track of where we are in the assembly process if an error occurs. The segment of code on lines 47 – 63 is responsible for detecting list, group start and stop tokens.

In each case when a token of the target entry type is found the index and a reference to the token are stored in local variables. On lines 66 – 75 we check for errors in the located start and stop indices and throw an exception if any are detected. The snippet of code that handles processing list tokens runs from line 75 to line 93. Similarly, the code to handle group tokens runs from line 94 to line 111.

This method is a little unique in how it handles its token scan. Notice that it doesn't aggregate list or group tokens after identifying the start token. Instead, it keeps track of the start and stop token of the given list or group and uses the index information as well as the start token to refactor the data representing the assembly source code.

A similar snippet of code, to that on lines 75 – 93, runs from line 94 to line 111. This code is almost identical to the previous snippet except that it's designed to handle group start and stop tokens. Take a moment to review the code in these two snippets. It should be familiar to you from our reviews of similar code during this chapter's discussion. Make sure you understand what the code does before you move on. At the outset of assembly step 8 we have a fully refactored representation of the assembly source code. In the upcoming section we'll review the remaining assembly steps that we're going to cover in this chapter, steps 9 – 11, and their associated pertinent methods.

## Main Method Details: AssemblerThumb.java - Step 9

Assembly step 9 marks the end of transformation steps and the beginning of four validation steps performed before we write out the final binary representation of the assembly program in step 13.

*Listing 18-16. AssemblerThumb.java Main Method Details 15*

```
001 public void PopulateOpCodeAndArgData(int step) throws
ExceptionRedefinitionOfLabel, ExceptionNoOpCodeFound,
ExceptionNoParentSymbolFound, ExceptionOpCodeAsArgument,
ExceptionMissingRequiredDirective {
002     if (eventHandler != null) {
003         eventHandler.PopulateOpCodeAndArgDataPre(step, this);
004     }
005
006     Logger.wrl("AssemblerThumb: PopulateOpCodeAndArgData");
007     boolean opCodeFound = false;
008     String opCodeName = null;
009     int opCodeIdx = -1;
010     int labelArgs = -1;
011     String lastLabel = null;
012     TokenLine lastLabelLine = null;
013     Symbol symbol = null;
014     boolean directiveFound = false;
015     boolean labelFound = false;
016
017     for (TokenLine line : asmDataTokened) {
018         lastLine = line;
019         if (eventHandler != null) {
020             eventHandler.PopulateOpCodeAndArgDataLoopPre(step, this,
line);
021         }
022
023         if (line.validLineEntry.index == 0 || line.validLineEntry.index
== 9) {
024             line.isEmpty = true;
025         }
026
027         opCodeFound = false;
028         opCodeName = null;
029         opCodeIdx = -1;
```

```
030     labelArgs = 0;  
031     directiveFound = false;  
032     labelFound = false;  
033  
034     for (Token token : line.payload) { ...
```

Part 1 of the AssemblerThumb class' PopulateOpCodeAndArgData method listing.

The operative function call for assembly step number 9 is to the class' PopulateOpCodeAndArgData method. The first part of the method, lines 1 – 32 shown previously, declares a number of important local variables. Let's take a look at the declarations starting on line 7. The variable `opCodeFound` is a Boolean flag used to indicate we have found an opcode token in the current token line. The following variable, `opCodeName`, is used to hold the name of the current opcode and the `opCodeIdx` integer holds the index of the opcode token in the current line.

The next few variables are associated with labels. Labels are another name for symbols or rather are how you declare a symbol of some kind, sometimes with an associated value. In this method we track the `labelArgs`, `lastLabel`, and `lastLabelLine` as we scan through the assembly source code's token lines. The subsequent variable, `symbol`, is used to create a new symbol to add to the symbol table. The remaining two variables, `directiveFound` and `labelFound`, are Boolean flags that are used to indicate certain tokens have been found, lines 14 – 15.

On line 17 we iterate over the token lines updating the `lastLine` class variable on line 18. On line 23 – 25 we check to see if the current token line is an empty line and if so, mark it as such. The segment of code on lines 27 – 32 prepares the local variables for the inner for loop and on line 34 we loop over the current token line's payload. We'll pick things up in the next listing with the first line of the inner for loop.

*Listing 18-17. AssemblerThumb.java Main Method Details 16*

```
035 lastToken = token;  
036 //MATCH REGISTERS
```

```

037 if (Utils.ArrayContainsString(JsonObjIsEntryTypes.NAME_REGISTERS,
token.type_name)) {
038     String regCode = token.source;
039     if (token.type_name.equals(JsonObjIsEntryTypes.NAME_REGISTERWB)) {
040         regCode = regCode.replace("!", " ");
041     }
042     regCode = regCode.replace(" ", " ");
043     regCode = regCode.replace(",", " ");
044
045     for (JsonObjIsRegister register : jsonObjIsRegisters.is_registers)
{
046         if (register.register_name.equals(regCode)) {
047             token.register = register;
048             token.isOpCodeArg = true;
049             break;
050         }
051     }
052 }
053
054 if (token.type_name.equals(JsonObjIsEntryTypes.NAME_OPCODE)) {
055     if (!opCodeFound) {
056         opCodeFound = true;
057         opCodeName = token.source;
058         opCodeIdx = token.index;
059     } else {
060         throw new ExceptionOpCodeAsArgument("Found OpCode token entry
where a sub-argument should be on line " + line.lineNumAbs + " with
argument index " + token.index);
061     }
062 } else if (token.type_name.equals(JsonObjIsEntryTypes.NAME_DIRECTIVE))
{
063     if (!directiveFound) {
064         directiveFound = true;
065     }
066 } else if (token.type_name.equals(JsonObjIsEntryTypes.NAME_START_LIST)
|| token.type_name.equals(JsonObjIsEntryTypes.NAME_START_GROUP) ||
token.type_name.equals(JsonObjIsEntryTypes.NAME_STOP_LIST) ||
token.type_name.equals(JsonObjIsEntryTypes.NAME_STOP_GROUP)) {
067     token.isOpCodeArg = true;
068
069 } else if (token.type_name.equals(JsonObjIsEntryTypes.NAME_LABEL)) {
070     if (token.index == 0) {
071         labelFound = true;

```

```

072     Token ltTmp = FindFirstDirective(line);
073     if (ltTmp == null || (ltTmp != null &&
074         Utils.ArrayContainsString(JsonObjIsDirectives.LABEL_DIRECTIVES,
075         ltTmp.source) == true)) {
076         if (ltTmp == null || (ltTmp != null &&
077             ltTmp.source.equals(JsonObjIsDirectives.NAME_EQU) == false)) {
078             lastLabel = token.source;
079             lastLabelLine = line;
080             if (symbols.symbols.containsKey(lastLabel)) {
081                 throw new ExceptionRedefinitionOfLabel("Found symbol '" +
082                     + lastLabel + "' redefined on line " + lastLabelLine.lineNumAbs + "
083                     originally defned on line " +
084                     (symbols.symbols.get(lastLabel)).lineNumAbs);
085             } else {
086                 Logger.wrl("AssemblerThumb: PopulateOpCodeAndArgData:
087                     Storing symbol with label '" + token.source + "' for line number " +
088                     line.lineNumAbs);
089             }
090             symbol = new Symbol();
091             symbol.line = line;
092             symbol.lineNumAbs = line.lineNumAbs;
093             symbol.addressBin = line.addressBin;
094             symbol.addressHex = line.addressHex;
095             symbol.addressInt = line.addressInt;
096             symbol.name = token.source;
097             symbol.token = token;
098             symbol.isLabel = true;
099             symbol.value = null;
100             symbols.symbols.put(token.source, symbol);
101         }
102     } else {
103         throw new ExceptionMissingRequiredDirective("Found symbol '" +
104             + lastLabel + "' found on line " + lastLabelLine.lineNumAbs + " is missing
105             required directive EQU that is expected when no OpCode is used.");
106     }
107 }
108 }
109 }
```

Part 2 of the AssemblerThumb class' PopulateOpCodeAndArgData method listing.

I'll take a moment to note that this method also updates the `lastToken` class field to keep both the `lastLine` and `lastToken` fields up to date with the current work being done. The segment of code on lines 37 – 52 is responsible for detecting if the current token is a register and for cleaning up a write back register, removing the ‘!’ character, lines 39 – 41. The segment of code also cleans the register’s assembly source code on lines 42 and 43, replacing spaces and commas. Also, note that on lines 45 – 51 we scan over the list of available registers and check if the current register token is in fact a valid register and mark it as such on lines 47 and 48.

The next if statement, lines 54 – 62, detects if the current token is an opcode and if so sets the `opCodeFound`, `opCodeName`, and `opCodeIdx` method variables on lines 56 – 58. Directives are detected on lines 63 – 65 and the `directiveFound` variable is set to true on line 64. Lines 66 – 69 detects when a token is an opcode argument and marks the token as such. As you can see from the review of the method thus far, its main responsibility is to mark tokens that are opcodes and opcode arguments, but it also handles the very important task of setting up entries in the symbol list when a label token is encountered. This is handled by the next block of code on lines 69 – 99. On line 70 there is an important check performed which indicates to us how the label token is being used.

We only want to add a new entry to the symbol table if the label token is the first token in the token line because labels can also appear as opcode arguments but if they are used as such, they won’t be the first token in the token line’s payload. The `labelFound` method variable is set to true on line 71 to indicate we’ve found a label token. There are a few checks we must perform before processing the token to make sure we’re not dealing with a directive argument. To do this we look for the first directive on the current token line, method line 72. Next, on lines 73 and 74 we check to see if we’re dealing with a label directive and if so, if the directive is `@EQU`, the main symbol processing code runs from lines 75 – 92 starting with updating the `lastLabel`, and `lastLabelLine` method variables on lines 75 – 76. In order to make sure that we don’t overwrite an existing symbol we perform a check on the symbol table, line 77.

If a symbol redefinition is not happening, we create a new symbol, lines 82 – 92, using the information that we have about the current line and token and insert it into the symbol table on line 92. We’re almost done with this

method's review, but we still have a few things left to cover. The next segment of code, listed subsequently, handles any sub-tokens that belong to the current token.

*Listing 18-18. AssemblerThumb.java Main Method Details 17*

```
100 //Process sub args
101 for (Token ltoken : token.payload) {
102     if (Utils.ArrayContainsString(JsonObjIsEntryTypes.NAME_REGISTERS,
103     ltoken.type_name)) {
104         String regCode = ltoken.source;
105         if
106             (ltoken.type_name.equals(JsonObjIsEntryTypes.NAME_REGISTERWB)) {
107                 regCode = regCode.replace("!", "");
108                 regCode = regCode.replace(" ", "");
109                 regCode = regCode.replace(",", "");
110                 for (JsonObjIsRegister register :
111                     jsonObjIsRegisters.is_registers) {
112                     if (register.register_name.equals(regCode)) {
113                         ltoken.register = register;
114                         ltoken.isOpCodeArg = true;
115                         break;
116                     }
117                 }
118             }
119             if (ltoken.type_name.equals(JsonObjIsEntryTypes.NAME_OPCODE)) {
120                 throw new ExceptionOpCodeAsArgument("Found OpCode token entry
where a sub-argument should be on line " + line.lineNumAbs + " with
argument index " + ltoken.index + " and parent argument index " +
token.index);
121
122             } else if
123                 (ltoken.type_name.equals(JsonObjIsEntryTypes.NAME_LABEL_REF) ||
124                 ltoken.type_name.equals(JsonObjIsEntryTypes.NAME_START_LIST) ||
125                 ltoken.type_name.equals(JsonObjIsEntryTypes.NAME_START_GROUP) ||
126                 ltoken.type_name.equals(JsonObjIsEntryTypes.NAME_STOP_LIST) ||
127                 ltoken.type_name.equals(JsonObjIsEntryTypes.NAME_STOP_GROUP)) {
128                     ltoken.isOpCodeArg = true;
129                 }
```

```
125 } ...
```

Part 3 of the AssemblerThumb class' PopulateOpCodeAndArgData method listing.

The case where a token has its own payload of tokens is called having sub-tokens, or child tokens. This case is handled on lines 101 – 124. The segment of code begins on line 101 with a loop to iterate over the current token's payload. The same register checks are performed on lines 103 to 116. This code is redundant, so I won't cover it again here. The remaining lines of code in the block, lines 119 to 123, are used to mark the sub-token as an opcode argument and to look for an incorrectly placed opcode token. That brings us to the last bit of code in the method, listed here.

*Listing 18-19. AssemblerThumb.java Main Method Details 18*

```
127 if (opCodeFound) {  
128     line.payloadOpCode = opCodeName;  
129     line.isLineOpCode = true;  
130     line.payloadLenArg = CountArgTokens(line.payload, 0);  
131     line.matchesOpCode = FindOpCodeMatches(line.payloadOpCode,  
line.payloadLenArg);  
132  
133     if (line.matchesOpCode == null || line.matchesOpCode.isEmpty()) {  
134         throw new ExceptionNoOpCodeFound("Could not find a matching  
opCode entry for name '" + line.payloadOpCode + "' with argument count " +  
line.payloadLenArg + " at line " + line.lineNumAbs + " with source text '"  
+ line.source.source + "'");  
135     }  
136 }  
137  
138 if (!opCodeFound && !directiveFound && labelFound) {  
139     line.isLineLabelDef = true;  
140 } ...
```

Part 4 of the AssemblerThumb class' PopulateOpCodeAndArgData method listing.

The last segment of code in the PopulateOpCodeAndArgData method runs from line 127 to 140. This code is responsible for updating the

current token line object to set the name of the opcode, the `isLineOpCode` Boolean flag, and the `payloadLenArg` field which is set via a call to the `CountArgTokens` support method. On line 131 a list of opcode matches are pulled from the current instruction set's data through a call to the `FindOpCodeMatches` support method.

Notice that on lines 133 – 135 we throw an exception if we haven't found an opcode match. We also set a `TokenLine` flag indicating if the line has a label token. That brings us to the conclusion of this method review and assembly step 9. In this step we've seen the token line objects updated and marked to indicate opcodes, opcode arguments, and label tokens.

## Main Method Details: AssemblerThumb.java - Step 10

The next step in the assembly process is to identify and mark directive arguments and token lines that are associated with area directives, `@CODE` and `@DATA`. This method is a large one, so I've broken it up into seven parts that we'll look at one at a time.

*Listing 18-20. AssemblerThumb.java Main Method Details 19*

```
001 public void PopulateDirectiveArgAndAreaData(int step) throws
ExceptionMissingRequiredDirective, ExceptionRedefinitionOfAreaDirective,
ExceptionNoDirectiveFound, ExceptionNoParentSymbolFound,
ExceptionMalformedEntryEndDirectiveSet, ExceptionNoAreaDirectiveFound,
ExceptionRedefinitionOfLabel, ExceptionNoSymbolFound {
002     if (eventHandler != null) {
003         eventHandler.PopulateDirectiveArgAndAreaDataPre(step, this);
004     }
005
006     Logger.wrl("AssemblerThumb: PopulateDirectiveAndArgData");
007     boolean dataDirectiveFound = false;
008     boolean directiveFound = false;
009     String directiveName = null;
010     int directiveIdx = -1;
011     int reqDirectiveCount = requiredDirectives.size() - 1;
012     List<String> reqDirectives = new ArrayList<>(requiredDirectives);
013
014     int lastEntry = -1;
015     int lastEnd = -1;
016     int lastArea = -1;
```

```

017     Token lastEntryToken = null;
018     Token lastEndToken = null;
019     Token lastAreaToken = null;
020     TokenLine lastEntryTokenLine = null;
021     TokenLine lastEndTokenLine = null;
022     TokenLine lastAreaTokenLine = null;
023
024     int lastCode = -1;
025     int lastData = -1;
026     int lastReadOnly = -1;
027     int lastReadWrite = -1;
028     int activeLineCount = 0;
029     AreaThumb tmpArea = null;
030
031     boolean foundTtl = false;
032     boolean foundSubt = false;
033     boolean foundArea = false;
034     boolean foundOrg = false;
035
036     String lastLabel = null;
037     TokenLine lastLabelLine = null;
038     Token lastLabelToken = null;
039     Symbol symbol = null;
040
041     for (TokenLine line : asmDataTokened) {
042         lastLine = line;
043         if (eventHandler != null) {
044             eventHandler.PopulateDirectiveArgAndAreaDataLoopPre(step,
this, line);
045         }
046
047         foundOrg = false;
048         dataDirectiveFound = false;
049         directiveFound = false;
050         directiveName = null;
051         directiveIdx = -1;
052
053         lastLabel = null;
054         lastLabelLine = null;
055         lastLabelToken = null;
056         symbol = null;
057
058         if (lastData != -1 && line.isLineOpCode) {

```

```
059         throw new ExceptionMalformedEntryEndDirectiveSet("Cannot have  
OpCode instructions when AREA type is DATA, found on line " +  
line.lineNumAbs + " with source " + line.source.source);  
060     }  
061  
062     for (Token token : line.payload) { ...
```

Part 1 of the AssemblerThumb class' PopulateDirectiveArgAndAreaData method listing.

The method begins with a large block of variable declarations that runs from line 7 to line 39. The first block of variables, lines 7 – 12, are used to track if certain directives exist in the tokenized data. The main purpose of this method is to detect and verify the required directives, and their arguments, exist. These directives are required to properly define an assembly source code file and the minimum required area and associated descriptive data directives. The second set of variables, lines 14 – 22, are all very similar and are used to track where certain directives are defined, if any.

Because the directives necessary to define an area of code have to exist on different lines, separated by assembly source code. This causes us to track the occurrence of individual directives across the entire tokenized source code. The structure of a code or data area is the occurrence of an @AREA directive followed by an @ENTRY directive. The entry directive marks the start of a block of opcodes, and data directives while the conclusion of the area is marked by an @END directive token.

The method variables, declared on lines 24 – 28, are used to indicate the line number of the last occurrence of a particular directive token. The activeLineCount variable is used to track the number of active lines in the tokenized data. Active lines are those that have opcodes, data directives, etc. Basically, any line of source code that affects the binary output of the assembler.

The next variable entry, tmpArea, is an instance of the AreaThumb class which is a thumb specific class used to track the lines of assembly code that are associated with a particular area directive. The variables on lines 31 – 34 are used to track the occurrence of certain directives. The remaining

variable declarations on lines 36 – 39 are used to track the occurrence of symbols by detecting label tokens in the token line scan. There is also a symbol variable, used to track symbols in the tokenized data.

In the same manner we've seen before we iterate over the tokenized data on line 41 and prepare local variables on lines 47 – 56. Notice that we don't reset all directive detection Boolean flags. Keep this in mind as we continue the review of this method. An exception is thrown in the code block on lines 58 – 60 if an opcode is detected in a data area and, on line 62, we scan the token payload of each TokenLine. We'll pick up the method review on line 63 in the subsequent listing.

*Listing 18-21. AssemblerThumb.java Main Method Details 20*

```
063 lastToken = token;
064 if (foundTtl &&
token.type_name.equals(JsonObjIsDirectives.NAME_DIRECTIVE_TYPE_STRING)) {
065     foundTtl = false;
066     assemblyTitle = token.source;
067
068 } else if (foundSubt &&
token.type_name.equals(JsonObjIsDirectives.NAME_DIRECTIVE_TYPE_STRING)) {
069     foundSubt = false;
070     assemblySubTitle = token.source;
071
072 } else if (foundArea &&
token.type_name.equals(JsonObjIsDirectives.NAME_DIRECTIVE_TYPE_STRING)) {
073     tmpArea.title = token.source;
074
075 } else if (token.type_name.equals(JsonObjIsEntryTypes.NAME_LABEL)) {
076     if (token.index == 0) {
077         lastLabel = token.source;
078         lastLabelLine = line;
079         lastLabelToken = token;
080     }
081 } else if (token.type_name.equals(JsonObjIsEntryTypes.NAME_LABEL_REF)) {
082     if (dataDirectiveFound == true && directiveFound == true &&
foundArea == true) {
083         token.isDirectiveArg = true;
084     }
```

```

085 } else if (token.type_name.equals(JsonObjIsEntryTypes.NAME_NUMBER)) {
086     if (directiveFound == true && foundArea == true) {
087         token.isDirectiveArg = true;
088     }
089
090     if (lastLabelToken != null && symbol != null) {
091         symbol.value = Utils.ParseNumberString(token.source);
092         symbols.symbols.put(lastLabel, symbol);
093         Logger.wrl("AssemblerThumb: PopulateDirectiveArgAndAreaData:
Storing symbol with label '" + lastLabel + "' for line number " +
lastLabelLine.lineNumAbs + " with value " + symbol.value);
094
095     lastLabel = null;
096     lastLabelToken = null;
097     lastLabelLine = null;
098     symbol = null;
099
100 } else if (foundOrg == true) {
101     asmStartLineNumber = Utils.ParseNumberString(token.source);
102     foundOrg = false;
103 }
104 } ...

```

Part 2 of the AssemblerThumb class' PopulateDirectiveArgAndAreaData method listing.

The second listing of the PopulateDirectiveArgAndAreaData method, shown previously, starts on line 63, with the first line of code from the inner for loop, the token scan loop. On lines 64 – 75 checks are performed to see if we are processing any directive arguments. For instance, on line 64, if we are in the @TTL directive and the current token is of type directive string, then we set the assemblyTitle variable and toggle off the foundTtl Boolean.

The same process is applied to the @SUBT and @AREA directives updating the assemblySubTitle and tmpArea.title respectively. Notice that on line 75 we check to see if we've encountered a label token. If the label token is the first token encountered in the token line, we update the lastLabel, lastLabelLine, and lastLabelToken variables, lines 77 –

79. Next, on lines 82 – 84 we check to see if we are looking at a label token after a data directive, regular directive, or area token has been encountered. In this case we set the `isDirectiveArg` flag of the current token to true, line 83.

The code on lines 86 – 103 is responsible for processing a number token. On lines 86 – 88, if a directive is found or an area is found then the current token is considered to be a directive argument and is marked as such, line 87. The next case we check when handling the number token, lines 91 – 98, is responsible for adding the value to the current symbol and updating the `symbols` Map with the new entry, lines 91 – 92. Following this, the method variables for tracking labels and symbols are reset on lines 95 – 98.

The last snippet of code in this section, lines 100 – 103, is responsible for handling the case when we encounter a number token, and we are processing tokens after an `@ORG` token has been encountered. We update the `asmStartLineNumber` field with the numeric value and toggle off the Boolean flag `foundOrg` on line 102. We'll pick up the review of this method on line 104 with the next else-if statement in the subsequent listing.

*Listing 18-22. AssemblerThumb.java Main Method Details 21*

```
104 } else if (token.type_name.equals(JsonObjIsEntryTypes.NAME_DIRECTIVE)) {  
105     if (!directiveFound) {  
106         directiveFound = true;  
107         directiveName = token.source;  
108         directiveIdx = token.index;  
109     }  
110  
111     if (reqDirectives.contains(token.source)) {  
112         reqDirectives.remove(token.source);  
113         reqDirectiveCount--;  
114     }  
115  
116     if (token.source.equals(JsonObjIsDirectives.NAME_TITLE)) {  
117         foundTtl = true;  
118         line.isEmpty = true;  
119     } else if (token.source.equals(JsonObjIsDirectives.NAME_SUB_TITLE)) {
```

```

121     foundSubt = true;
122     line.isEmpty = true;
123
124 } else if (token.source.equals(JsonObjIsDirectives.NAME_ORG)) {
125     foundOrg = true;
126     line.isEmpty = true;
127
128 } else if (token.source.equals(JsonObjIsDirectives.NAME_EQU)) {
129     line.isEmpty = true;
130     if (symbols.symbols.containsKey(lastLabel)) {
131         throw new ExceptionRedefinitionOfLabel("Found symbol '" +
lastLabel + "' redefined on line " + lastLabelLine.lineNumAbs +
originally defned on line " +
(symbols.symbols.get(lastLabel)).lineNumAbs);
132     }
133     symbol = new Symbol();
134     symbol.line = line;
135     symbol.lineNumAbs = line.lineNumAbs;
136     symbol.addressBin = line.addressBin;
137     symbol.addressHex = line.addressHex;
138     symbol.addressInt = line.addressInt;
139     symbol.name = lastLabel;
140     symbol.token = lastLabelToken;
141     symbol.isStaticValue = true;
142
143 } ...

```

Part 3 of the AssemblerThumb class' PopulateDirectiveArgAndAreaData method listing.

Part 3 of the method review is shown in the previous listing. Starting on line 104, if the current token is a directive token, then the code block starting on line 105 executes. This is a large block of code that extends into the next listing. We'll start looking at the first half of it, line 105 – 142, shown previously. On lines 105 – 109, if we haven't already encountered a directive token, we set local variable `directiveFound` to true, update the `directiveName`, and set the index the directive was found at, on line 108.

Next, on lines 111 – 114, we check to see if we have encountered a required directive and if so, we remove it from the list of required tokens and

the required token count. This code tracks if all required tokens have indeed been found while scanning the tokenized assembly source code. Starting with line 116 we have to check the different directive types and process the associated Boolean flags accordingly. Recall, that these flags are checked in previous sections of this method to see if newly encountered tokens are directive arguments and how to process them.

If the current token is a title directive the `foundTtl` flag is set to true and the line's `isLineEmpty` flag is also set to true. The same process is applied to the subtitle and org directives on lines 120 – 128. The `@EQU` directive is handled in the block of code on lines 128 to 143. This line is also marked as empty because it's a symbol definition line and doesn't end up in the binary version of the program. The symbol is created, and its attributes are set based on the current token line and the last label, local variables. The line that's being processed in this case is as follows.

```
test      @EQU      2
```

The first token in the line is a label, the second is the `@EQU` directive, which we're processing here, and the third is a number token. Recall from previous parts of this method that the number token is used to set a symbol value in the case the `@EQU` directive has been encountered. This method is complex because it's processing assembly code features like this. The next listing for this method picks up on line 143 with the remainder of the directive cases to handle. Let's take a look.

*Listing 18-23. AssemblerThumb.java Main Method Details 22*

```
143 } else if (token.source.equals(JsonObjIsDirectives.NAME_AREA)) {  
144     line.isLineEmpty = true;  
145     if (lastArea == -1) {  
146         foundArea = true;  
147         lastArea = line.lineNumAbs;  
148         lastAreaToken = token;  
149         lastAreaTokenLine = line;  
150         tmpArea = new AreaThumb();  
151         tmpArea.area = lastAreaToken;  
152         tmpArea.areaLine = lastAreaTokenLine;  
153         tmpArea.lineNumArea = lastArea;
```

```

154     } else {
155         throw new ExceptionRedefinitionOfAreaDirective("Redefinition
of AREA directive found on line " + line.lineNumAbs + " with source " +
line.source.source);
156     }
157 } else if (token.source.equals(JsonObjIsDirectives.NAME_CODE)) {
158     line.isEmpty = true;
159     if (lastCode == -1) {
160         lastCode = line.lineNumAbs;
161     }
162     tmpArea.isCode = true;
163     tmpArea.isReadOnly = true;
164     if (lastData != -1 && lastData == lastCode) {
165         throw new ExceptionMalformedEntryEndDirectiveSet("Cannot set
AREA type to CODE when type is DATA, found on line " + line.lineNumAbs + "
with source " + line.source.source);
166     }
167 } else if (token.source.equals(JsonObjIsDirectives.NAME_DATA)) {
168     line.isEmpty = true;
169     if (lastData == -1) {
170         lastData = line.lineNumAbs;
171     }
172     tmpArea.isData = true;
173     tmpArea.isReadWrite = true;
174     if (lastCode != -1 && lastCode == lastData) {
175         throw new ExceptionMalformedEntryEndDirectiveSet("Cannot set
AREA type to DATA when type is CODE, found on line " + line.lineNumAbs + "
with source " + line.source.source);
176     }
177 } else if (token.source.equals(JsonObjIsDirectives.NAME_READONLY)) {
178     line.isEmpty = true;
179     if (lastReadOnly == -1) {
180         lastReadOnly = line.lineNumAbs;
181     }
182     tmpArea.isReadOnly = true;
183     tmpArea.isReadWrite = false;
184     if (lastReadOnly != -1 && lastReadOnly == lastReadWrite) {
185         throw new ExceptionMalformedEntryEndDirectiveSet("Cannot set
AREA type to READONLY when type is READWRITE, found on line " +
line.lineNumAbs + " with source " + line.source.source);
186     }
187 } ...

```

## Part 4 of the AssemblerThumb class' PopulateDirectiveArgAndAreaData method listing.

We pick things up on line 143 with an else-if statement that detects if the current token is an @AREA directive token. The line is marked empty because this isn't a directive that appears in the binary representation of the program. On line 145 we check to see if an area directive has already been defined or not. If not, the code on lines 146 to 153 executes. If so, we throw an exception because we shouldn't encounter a new area definition inside an already existing area definition.

Next up, the code on lines 146 – 149 marks the area as found and records the line, token, and absolute line number of the occurrence. On lines 157 – 167 we handle the case that the current token is an instance of the @CODE directive. Line 158 sets the line's isLineEmpty field to true, then on lines 159 – 161 we check to see if the code directive has already been encountered. Because this attribute only occurs after an area directive, we use it to set the fields of the current area on lines 162 and 163.

The same process we just reviewed is also applied to the @DATA directive. This directive is used in almost the same way as the code directive so the code to handle it is very similar. Take a moment to look over the code on lines 167 – 177. The last directive checked in this code block is the @READONLY directive, processed on lines 177 – 187. This directive is also handled in much the same way as the previous two directives we just reviewed. Again, take a moment to look over the code and make sure you understand it before moving onto the next listing.

*Listing 18-24. AssemblerThumb.java Main Method Details 23*

```
187     } else if (token.source.equals(JsonObjIsDirectives.NAME_READWRITE)) {
188         line.isLineEmpty = true;
189         if (lastReadWrite == -1) {
190             lastReadWrite = line.lineNumAbs;
191         }
192         tmpArea.isReadWrite = true;
193         tmpArea.isReadOnly = false;
```

```

194     if (lastReadWrite != -1 && lastReadWrite == lastReadOnly) {
195         throw new ExceptionMalformedEntryEndDirectiveSet("Cannot set
AREA type to READWRITE when type is READONLY, found on line " +
line.lineNumAbs + " with source " + line.source.source);
196     }
197 } else if (token.source.equals(JsonObjIsDirectives.NAME_ENTRY)) {
198     line.isEmpty = true;
199     if (lastArea == -1) {
200         throw new ExceptionNoAreaDirectiveFound("Could not find AREA
directive before ENTRY directive on line " + line.lineNumAbs + " with
source " + line.source.source);
201     } else if (lastEntry == -1) {
202         lastEntry = line.lineNumAbs;
203         lastEntryToken = token;
204         lastEntryTokenLine = line;
205         tmpArea.entry = lastEntryToken;
206         tmpArea.entryLine = lastEntryTokenLine;
207         tmpArea.lineNumEntry = lastEntry;
208     } else {
209         throw new ExceptionMalformedEntryEndDirectiveSet("Found
multiple ENTRY directives with a new entry on line " + line.lineNumAbs + "
with source " + line.source.source);
210     }
211 } else if (token.source.equals(JsonObjIsDirectives.NAME_END)) {
212     line.isEmpty = true;
213     if (lastArea == -1) {
214         throw new ExceptionNoAreaDirectiveFound("Could not find AREA
directive before ENTRY directive on line " + line.lineNumAbs + " with
source " + line.source.source);
215     } else if (lastEntry == -1) {
216         throw new ExceptionMalformedEntryEndDirectiveSet("Could not
find END directive before new ENTRY directive on line " + line.lineNumAbs +
" with source " + line.source.source);
217     } else if (lastEnd == -1) {
218         lastEnd = line.lineNumAbs;
219         lastEndToken = token;
220         lastEndTokenLine = line;
221         tmpArea.end = lastEndToken;
222         tmpArea.endLine = lastEndTokenLine;
223         tmpArea.lineNumEnd = lastEnd;
224     }
225     if (tmpArea.isData) {
226         if (areaThumbData == null) {

```

```

227             areaThumbData = tmpArea;
228         } else {
229             throw new ExceptionMalformedEntryEndDirectiveSet("Can
only define one DATA AREA, second definition on line " +
tmpArea.lineNumArea + " with source " + tmpArea.areaLine.source.source);
230         }
231     } else {
232         if (areaThumbCode == null) {
233             areaThumbCode = tmpArea;
234         } else {
235             throw new ExceptionMalformedEntryEndDirectiveSet("Can
only define one CODE AREA, second definition on line " +
tmpArea.lineNumArea + " with source " + tmpArea.areaLine.source.source);
236         }
237     }
238
239     if (lastEnd <= lastEntry) {
240         throw new ExceptionMalformedEntryEndDirectiveSet("Could
not find END directive before new ENTRY directive on line " +
line.lineNumAbs + " with source " + line.source.source);
241     }
242
243     tmpArea = null;
244     lastArea = -1;
245     lastAreaToken = null;
246     lastAreaTokenLine = null;
247     lastEntry = -1;
248     lastEntryToken = null;
249     lastEntryTokenLine = null;
250     lastEnd = -1;
251     lastEndToken = null;
252     lastEndTokenLine = null;
253     lastCode = -1;
254     lastData = -1;
255     lastReadOnly = -1;
256     lastReadWrite = -1;
257     foundArea = false;
258 } else {
259     throw new ExceptionMalformedEntryEndDirectiveSet("Could not
find END directive before new ENTRY directive on line " + line.lineNumAbs
+ " with source " + line.source.source);
260 }
```

```

261     } else if (token.source.equals(JsonObjIsDirectives.NAME_DCHW) ||
262     token.source.equals(JsonObjIsDirectives.NAME_DCB) ||
263     token.source.equals(JsonObjIsDirectives.NAME_FLPDCHW) ||
264     token.source.equals(JsonObjIsDirectives.NAME_DCWBF) ||
265     token.source.equals(JsonObjIsDirectives.NAME_DCWBS)) {
266         if (lastArea == -1 || tmpArea == null) {
267             throw new ExceptionNoAreaDirectiveFound("Could not find AREA
directive before directive '" + token.source + "' on line " +
line.lineNumAbs + " with source " + line.source.source);
268         } else {
269             dataDirectiveFound = true;
270             token.isDirective = true;
271             if (!directiveFound) {
272                 directiveFound = true;
273                 directiveName = token.source;
274                 directiveIdx = token.index;
275             }
276             if (lastLabel != null) {
277                 symbol = symbols.symbols.get(lastLabel);
278                 if (symbol == null) {
279                     throw new ExceptionNoSymbolFound("Could not find a
symbol with name " + lastLabel + " at line number " + line.lineNumAbs + "
with source " + line.source.source);
280                 }
281             }
282         }
283     if (directiveFound) {
284         line.payloadDirective = directiveName;
285         line.isLineDirective = true;
286
287         //Allow for Directive args and OpCode args since we can't define
two Entry Types that are identical but have different categories
288         line.payloadLenArg = CountArgTokens(line.payload, 0,
JsonObjIsEntryTypes.NAME_CAT_ARG_DIRECTIVE, false) +
CountArgTokens(line.payload, 0, jsonObjIsEntryTypes.NAME_CAT_ARG_OPCODE,
false);
289         line.matchesDirective = FindDirectiveMatches(line.payloadDirective,
line.payloadLenArg);
290     } ...

```

Part 5 of the AssemblerThumb class' PopulateDirectiveArgAndAreaData method listing.

The next listing from the PopulateDirectiveArgAndAreaData method, shown previously, is responsible for handling the remaining directives that we haven't encountered yet. We start things off on line 187 with a check to see if we've encountered the @READWRITE directive. The line is marked as empty on line 188 and the local method variable, lastReadWrite, is updated on lines 189 – 191 to indicate the line number the directive was encountered on.

Because this directive is used to set the attributes of the current area, even though these attributes are not currently being enforced, on lines 192 and 193. The remaining lines of code in this if statement, lines 194 – 196, check to see if there is a redefinition of this attribute and throw an exception if one is found. The next directive check runs from line 211 to line 261 and is responsible for determining if we encounter an @END directive in our scan.

This case has special significance because the @END directive is the closing directive for an area of source code. For instance, the GenAsm assembler supports two area types, code and data. Each area declared has a starting and ending point. The starting point is designated by the @ENTRY directive and the ending point is designated by the @END directive, as you may have guessed. Let's see how we handle the closure of a defined area.

On line 212, the isEmpty, Boolean flag is set to true to indicate that this line of assembly source code is not present in the final binary representation of the program. The checks on line 213 to 217 are there to ensure the area, entry and end directives are all properly declared and in the correct order. Next up, on lines 217 – 258 we handle the end directive. The local variables indicating the line number, token line, and token associated with the end directive are updated on lines 218 – 220 while the tmpArea variable is updated with information about the end directive on lines 221 – 223.

Next, on lines 225 – 237, depending on the type of area directive found, code or data, we perform some null checks before setting the

`areaThumbCode` or `areaThumbData` local variables based on the `tmpArea` variable. There is a special check on lines 239 – 241 to ensure the area's entry and end token are properly ordered. Lastly, the local method variables for tracking the definition of an area and its associated entry and end tokens are all reset on lines 243 – 257.

The last else-if statement on line 261 – 280 is responsible for handling all data directives. Data directives are one of the few directives that end up in the final binary representation of the program. Before we do any real work in this branch we check first to see if the area this directive belongs to is defined and throw an exception otherwise.

The `dataDirectiveFound` Boolean is toggled on line 265 and the token is marked as being a directive token on line 266. I should mention that the `isDirective` field of the `Token` class is reserved for directives that alter the resulting binary representation of the program. The local variables that are used to indicate that a directive has been found are set on lines 267 – 271. There is a specific case we need to check here. If a label has been encountered prior to the data directive token, then we need to check that the symbol is defined and throw an exception otherwise, lines 274 – 277.

I'll mention that we're still working with the inner for loop, the token scanning loop, at this point in the method. The last if statement of the inner for loop runs from line 283 – 290. If a directive has been found after scanning the current line's tokens, then we set the `TokenLine`'s `payloadDirective` and `isLineDirective` flags on lines 284 – 285. A couple of important lookups are performed on lines 288 and 289. The first such lookup is used to calculate the total number of arguments, both opcode and directive, that exist on the current token line.

The second lookup ties into a support method we reviewed not too long ago. The call to the `FindDirectiveMatches` method is used to populate the line's `matchesDirective` field with all matching directives that have the same name and argument count. That brings us to the end of this method review section and the conclusion of the inner for loop's review. We'll pick things up in the next listing for this method on line 293. This segment of code belongs to the outer for loop, the line scanning for loop.

*Listing 18-25. AssemblerThumb.java Main Method Details 24*

```

293 if (directiveFound) {
294     if (line.matchesDirective == null ||
line.matchesDirective.isEmpty()) {
295         throw new ExceptionNoDirectiveFound("Could not find a matching
directive entry for name '" + line.payloadDirective + "' with argument
count " + line.payloadLenArg + " at line " + line.lineNumAbs + " with
source text '" + line.source.source + "'");
296     }
297 }
298 ...

```

Part 6 of the AssemblerThumb class' PopulateDirectiveArgAndAreaData method listing.

This is a very short code listing. At the end of the token line scan, outer for loop, there exists an if statement to ensure that if a directive has been found that it has at least one matching directive in the instruction set's directive definitions. If not, we exit by throwing an exception indicating something has gone wrong, line 295. We're not out of the woods just yet. I know this method review has been arduous but it's vitally important to the assembler and as such deserves our time and attention.

*Listing 18-26. AssemblerThumb.java Main Method Details 25*

```

304 if (reqDirectiveCount > 0) {
305     String lmissing = "";
306     for (int i = 0; i < reqDirectives.size(); i++) {
307         lmissing += reqDirectives.get(i);
308         if (i < reqDirectives.size() - 1) {
309             lmissing += ",";
310         }
311     }
312     throw new ExceptionMissingRequiredDirective("Could not find
required directive in the source file, '" + lmissing + "'");
313 }
314
315 CleanAreas();
316 CleanSymbols();

```

Part 7 of the AssemblerThumb class' PopulateDirectiveArgAndAreaData method listing.

Listing part 7 of the method review, shown previously, is short and sweet. This block of code occurs immediately after the end of the outer for loop, the line scanning for loop. This code is responsible for reporting any errors regarding the directives required to properly format GenAsm assembly source code. If the local variable `reqDirectiveCount` is not zero, then we scan through the list of required directives and build a string representation of the missing entries.

The information is used to report the exception and exit the method, line 312. The last two lines of code, lines 315 – 316 are used to call the `CleanAreas` and `CleanSymbols` methods. I'm leaving the review of these methods up to you. The `CleanAreas` method is rather direct, it's used to clean and associate `TokenLines` with their associated area directive. Similarly, the `CleanSymbols` method is used to clean any bad symbols out of the symbol list. Take a moment to look over these methods on your own. Make sure you understand them before moving on to the next chapter in this text.

## Chapter Conclusion

That brings us to the conclusion of this chapter. We've managed to review assembler steps 1 – 10 and have taken care of a number of important validation and refactoring steps. The remaining steps in the assembly process, steps 11 – 13, are used for a deep validation of the tokenized assembly source code's lines before the final binary representation of each `TokenLine` is generated in step 13. Let's take a look at the material we've covered in this chapter.

- **STEP 1:** Process `JsonObjIsSet`'s file entries and load then parse the JSON object data.  
Operative Method: `LoadAndParseJsonObjData`

- **STEP 2:** Link loaded JSON object data.  
Operative Method: LinkJsonObjData
- **STEP 3:** Load and lexerize the assembly source file.  
Operative Method: LexerizeAssemblySource
- **STEP 4:** Tokenize the lexerized artifacts.  
Operative Method: TokenizeLexerArtifacts
- **STEP 5:** Validate token lines.  
Operative Method: ValidateTokenizedLines
- **STEP 6:** Combine comment tokens as children of the initial comment token.  
Operative Method: CollapseCommentTokens
- **STEP 7:** Expand register ranges into individual register entries.  
Operative Method: ExpandRegisterRangeTokens
- **STEP 8:** Combine list and group tokens as children of the initial list or group token.  
Operative Method: CollapseListAndGroupTokens
- **STEP 9:** Mark OpCode, OpCode argument, and register tokens.  
Operative Method: PopulateOpCodeAndArgData
- **STEP 10:** Mark directive and directive argument tokens, create area-based line lists with hex numbering.  
Operative Method: PopulateDirectiveArgAndAreaData

We still have a bit more code to review to complete this class review. In Chapter 19 we'll look at deep line validation, steps 11 and 12, and the binary representation generated in step 13. Once we wrap up the `AssemblerThumb` class' main method review we'll take a look at the `LinkerThumb` class, the last class for us to review, and then we'll take a look at the included example programs.

# **Chapter 19: The Assembler Part 3**

Welcome to part 3 of the assembler review. In part 1 we managed to review the overall process, listing each step taken by the assembler and its associated operative main method, class fields, headers, pertinent class methods, and the `AssemblerThumb` class' support methods. No small task. In part 2 we reviewed assembler steps 1 – 10. This took us through a variety of main methods with varying responsibilities including loading data, parsing and processing the data, simple line validation, and data structure refactoring with regard to the handling of comment, list, and group `Tokens`.

In this chapter we'll finish our review of the `AssemblerThumb` class, specifically the main method review, with a detailed discussion of the remaining assembler steps, steps 11 – 13.

## **Main Method Details: `AssemblerThumb.java` - Step 11**

Welcome to assembly step 11. At this point in our review of the `AssemblyThumb` class we've completed all the data structure refactoring, simple validations, and code / data area preparations which will be picked up by the linker, as we'll see in just a little bit. In this assembly step, step 11 of

13, the operative method called is the `ValidateOpCodeLines` method, listed as follows. Hang in there we're almost through this main method review. Let's jump into some code!

*Listing 19-1. AssemblerThumb.java Main Method Details 1*

```
01 public void ValidateOpCodeLines(int step) throws
ExceptionNoOpCodeFound, ExceptionNoOpCodeLineFound {
02     if (eventHandler != null) {
03         eventHandler.ValidateOpCodeLinesPre(step, this);
04     }
05
06     Logger.wrl("AssemblerThumb: ValidateOpCodeLines");
07     boolean opCodeFound = false;
08     String opCodeName = null;
09     Token opCodeToken = null;
10     int opCodeIdx = -1;
11     List<Token> args = null;
12     JsonObjectIsOpCode opCode = null;
13
14     for (TokenLine line : asmDataTokened) {
15         lastLine = line;
16         if (eventHandler != null) {
17             eventHandler.ValidateOpCodeLinesLoopPre(step, this, line);
18         }
19
20         if (line.isLineOpCode) {
21             opCodeFound = false;
22             opCodeName = null;
23             opCodeToken = null;
24             opCodeIdx = -1;
25             args = null;
26
27             for (Token token : line.payload) {
28                 lastToken = token;
29                 if (!opCodeFound) {
30                     if
31 (token.type_name.equals(JsonObjIsEntryTypes.NAME_OPCODE)) {
32                         opCodeFound = true;
33                         opCodeName = token.source;
34                         opCodeToken = token;
35                         opCodeIdx = token.index;
```

```

35                 args = new ArrayList<>();
36             }
37         } else {
38             args.add(token);
39         }
40
41         //Adjust if label is defined or referenced
42         if (opCodeFound &&
(token.type_name.equals(JsonObjIsEntryTypes.NAME_LABEL_REF))) {
43             token.isLabelRef = true;
44             token.isLabel = false;
45         }
46     }
47
48     if (opCodeFound && args != null && args.size() > 0) {
49         opCode = FindOpCodeArgMatches(line.matchesOpCode, args,
opCodeToken);
50         line.matchesOpCode.clear();
51         line.matchesOpCode.add(opCode);
52     }
53
54     if ((line.matchesOpCode != null && line.matchesOpCode.size() >
1) || Utils.IsEmpty(line.matchesOpCode)) {
55         throw new ExceptionNoOpCodeFound("Could not find unique
matching opCode entry for opCode '" + opCodeName + "' and line number " +
line.lineNumAbs + " with source '" + line.source.source + "'");
56     }
57 }
58
59     if (eventHandler != null) {
60         eventHandler.ValidateOpCodeLinesLoopPost(step, this, line);
61     }
62 }
63
64     for (String key : symbols.symbols.keySet()) {
65         Symbol symbol = symbols.symbols.get(key);
66         TokenLine line = asmDataTokened.get(symbol.lineNumAbs);
67         TokenLine tmpLine = null;
68         if (Utils.ArrayContainsInt(JsonObjIsValidLines.LINES_LABEL_EMPTY,
line.validLineEntry.index)) {
69             tmpLine = FindNextOpCodeLine(symbol.lineNumAbs, key);
70             symbol.addressInt = tmpLine.addressInt;
71             symbol.addressBin = tmpLine.addressBin;

```

```

72     symbol.addressHex = tmpLine.addressHex;
73     symbol.lineNumActive = tmpLine.lineNumActive;
74     symbol.isEmptyLineLabel = true;
75     Logger.wrl("Adjusting symbol, '" + symbol.name + "', line
number from " + symbol.lineNumAbs + " to " + symbol.addressInt + " due to
symbol marking an empty line");
76   } else {
77     symbol.addressInt = line.addressInt;
78     symbol.addressBin = line.addressBin;
79     symbol.addressHex = line.addressHex;
80     symbol.lineNumActive = line.lineNumActive;
81   }
82 }
83
84 if (eventHandler != null) {
85   eventHandler.ValidateOpCodeLinesPost(step, this);
86 }
87 }
```

A listing of the AssemblerThumb class' ValidateOpCodeLines method listing.

The ValidateOpCodeLines method, shown in the previous listing, is responsible for making sure that token lines, marked as having an opcode, are valid. What this means is that we want to ensure that the opcode and its arguments are valid when compared to the JSON data files that define the instruction set. Method variables are declared on lines 7 – 12. The first entry, opCodeFound, is a Boolean flag that indicates if we've encountered an opcode line during our scan.

The opCodeName variable is used to track the name of the current opcode if we've found one. The next variable is used to track the token where the opcode was found and the opCodeIdx variable is used to track the index of that token. The next variable, args, holds the opcode's associated arguments and lastly opCode, line 12, is a holder class that represents the matching opcode in the instruction set's data files.

We iterate over the assembly token line objects on line 14. Notice that we update the lastLine class field to keep track of the token line we're processing. On lines 21 – 56 we handle the case that the current token line is

marked as having an opcode. The method variables are reset on lines 21 – 25. The inner for loop starts on line 27. We iterate over the current token line's payload.

The `lastToken` class field is updated on line 28 and if we haven't encountered an opcode token yet then we check if the current token is an opcode token. If so, the `opCodeFound` Boolean flag is set to true and the remaining method variables are updated based on the current token. The `args` list is initialized on line 35. On line 38 we add the current token to the opcode's `args` list. Note that we can do this because the opcode token always precedes its arguments. On lines 42 – 45, if an opcode has been found and if the current token is a label reference, we update the token line's `isLabelRef` and `isLabel` fields.

Next, on lines 48 – 52, outside the inner for loop, we look up the opcode's arguments that match the known arguments in the current line. Lastly, on lines 64 – 82, we scan the symbol list to make sure that the symbols reference the correct line and make any necessary adjustments to reset the address and line number fields of each symbol. That brings us to the conclusion of this assembly step review. In the next section we'll take a look at assembly step 12 and its operative method.

## Main Method Details: AssemblerThumb.java - Step 12

Assembly step 12 sees us continue our validation of the tokenized directive lines at this stage in the process. The operative method in this step of the assembly process is the `ValidateDirectiveLines` method. Let's take a look at how the method works.

*Listing 19-2. AssemblerThumb.java Main Method Details 2*

```
01 public void ValidateDirectiveLines(int step) throws
ExceptionNoDirectiveFound {
02     if (eventHandler != null) {
03         eventHandler.ValidateDirectiveLinesPre(step, this);
04     }
05
06     Logger.wrl("AssemblerThumb: ValidateDirectiveLines");
07     boolean directiveFound = false;
08     String directiveName = null;
```

```

09     Token directiveToken = null;
10     int directiveIdx = -1;
11     List<Token> args = null;
12     JsonObjectIsDirective directive = null;
13
14     for (TokenLine line : asmDataTokened) {
15         lastLine = line;
16         if (eventHandler != null) {
17             eventHandler.ValidateDirectiveLinesLoopPre(step, this, line);
18         }
19
20         if (line.isLineDirective) {
21             directiveFound = false;
22             directiveName = null;
23             directiveToken = null;
24             directiveIdx = -1;
25             args = null;
26
27             for (Token token : line.payload) {
28                 lastToken = token;
29                 if (!directiveFound) {
30                     if
31 (token.type_name.equals(JsonObjectIsEntryTypes.NAME_DIRECTIVE)) {
32                         directiveFound = true;
33                         directiveName = token.source;
34                         directiveToken = token;
35                         directiveIdx = token.index;
36                         args = new ArrayList<>();
37                     }
38                 } else {
39                     args.add(token);
40                 }
41
42                 if (directiveFound && args != null && args.size() > 0) {
43                     directive = FindDirectiveArgMatches(line.matchesDirective,
44 args, directiveToken);
45                     line.matchesDirective.clear();
46                     line.matchesDirective.add(directive);
47                 }

```

```

48         if ((line.matchesDirective != null &&
49             line.matchesDirective.size() > 1) ||
50             Utils.IsEmpty(line.matchesDirective)) {
51             throw new ExceptionNoDirectiveFound("Could not find unique
52             matching directive entry for directive '" + directiveName + "' and line
53             number " + line.lineNumAbs + " with source '" + line.source.source + "'");
54         }
55     }
56 }
57
58 if (eventHandler != null) {
59     eventHandler.ValidateDirectiveLinesLoopPost(step, this, line);
60 }
61 }

```

A listing of the AssemblerThumb class' ValidateOpCodeLines method.

The ValidateDirectiveLines method is similar to the ValidateOpCodeLines method we've just reviewed. On lines 7 – 12 the method variables are declared. The first entry is the directiveFound Boolean which indicates if a directive has been found. The second entry, directiveName, tracks the name of the current directive if any. Following this field entry is the directiveToken field which references the token where we encountered the directive. The last two entries are the args list and directive holder class, lines 11 – 12.

Next, we iterate over the list of token lines on line 14. The lastLine class field is updated on line 15 and a check to see if the current line has a directive token is performed on line 20. The method variables are updated on lines 21 – 25. The inner for loop, which is used to iterate over the tokens in the current token line's payload, starts on line 27.

The class' lastToken field is updated on line 28 and if a directive hasn't been encountered, line 29, we set the directiveFound variable to true and update the directiveName variable on line 32. Next, on lines 33 –

34 the token where the directive was found, and the index of the token are set. The `args` list is initialized on line 35. If a directive has been found the token is added to the `args` variable, line 38.

The if statement on line 42 checks to see if a directive has been found and if it has arguments defined. If it does have defined arguments, then we look up the directive arguments on line 43 with a call to the `FindDirectiveArgMatches` support method. The results are stored in the `tokenLine` object on lines 44 and 45. The subsequent if statement is there to check if any directive matches were found and throw an exception if not. That brings us to the conclusion of the current method review. Before moving on take a moment to understand what exceptions are thrown by this method and why. In the next section we'll take a look at the class' last main method to review.

## Main Method Details: AssemblerThumb.java - Step 13

The last step in the assembly process is to build out the binary representation of active token lines. Recall that we consider an active token line one that exists in the final binary representation of the program. There are few operative methods used in this assembly step. The first such method is the `BuildBinLines` method. This method is responsible for iterating over the token lines in code and data areas and calls the `BuildBinOpCode` and `BuildBinDirective` methods to create the binary representation of the current line.

*Listing 19-3. AssemblerThumb.java Main Method Details 3*

```
01 public void BuildBinLines(int step, List<TokenLine> areaLines,
AreaThumb area) throws ExceptionOpCodeAsArgument, ExceptionNoSymbolFound,
ExceptionUnexpectedTokenWithSubArguments, ExceptionNumberInvalidShift,
ExceptionNumberOutOfRange, ExceptionNoNumberRangeFound,
ExceptionUnexpectedTokenType, ExceptionInvalidEntry, ExceptionInvalidArea,
ExceptionInvalidAssemblyLine, ExceptionDirectiveArgNotSupported,
ExceptionMissingDataDirective, ExceptionNoSymbolValueFound {
02     if (eventHandler != null) {
03         eventHandler.BuildBinLinesPre(step, this, areaLines, area);
04     }
05
06     if (area.isCode) {
```

```

07     for (TokenLine line : areaLines) {
08         lastLine = line;
09         BuildBinOpCode(step, line);
10         BuildBinDirective(step, line);
11     }
12 } else if (area.isData) {
13     for (TokenLine line : areaLines) {
14         lastLine = line;
15         BuildBinDirective(step, line);
16     }
17 } else {
18     throw new ExceptionInvalidArea("Found an invalid area entry at
line number " + area.areaLine + " width code: " + area.isCode + " and
data: " + area.isData);
19 }
20
21 if (eventHandler != null) {
22     eventHandler.BuildBinLinesPost(step, this);
23 }
24 }
```

A listing of the AssemblerThumb class' BuildBinLines method.

The first pertinent method of this assembly step is the BuildBinLines method, shown in the previous listing. At this point in the assembly process, we've finished refactoring and validating the assembly token lines and we're ready to start building the binary representation of each line. This prepares the tokenized data for use by the linker. We'll spend more time on this topic soon when we review the linker class. On line 6 of the method, we check to see if the area argument provided is a code area.

If so, on lines 7 – 11 we iterate over the area's lines and call the BuildBinOpCode and BuildBinDirective methods after updating the lastLine class field, lines 8 – 10. Can you think of a reason why we might want to process both opcode and directive instructions here? Remember that we're working on the code area and as such we have to handle both opcode and directive instructions.

The case where the provided area is a data area is handled on lines 12 – 17. We iterate over the data area's lines, lines 13 – 16, updating the `lastLine` class field and calling the `BuildBinDirective` method for each line of the area. Lastly, on lines 17 – 19, we throw an exception in the case that the provided area isn't marked as a code or data area. That wraps up our review of the `BuildBinLines` method.

We've worked our way through a lot of code in this chapter. We're almost finished with the main method review but, we have a little bit more code to look over. Next up, let's take a look at one of the remaining pertinent methods used in step 13 of the assembly process. The next method is fairly large, so I'll break it up into a few parts. Part 1 of the method is listed as follows.

#### *Listing 19-4. AssemblerThumb.java Main Method Details 4*

```
001 public void BuildBinDirective(int step, TokenLine line) throws
ExceptionOpCodeAsArgument, ExceptionNoSymbolFound,
ExceptionUnexpectedTokenWithSubArguments, ExceptionNumberInvalidShift,
ExceptionNumberOutOfRange, ExceptionNoNumberRangeFound,
ExceptionUnexpectedTokenType, ExceptionInvalidEntry,
ExceptionInvalidAssemblyLine, ExceptionDirectiveArgNotSupported,
ExceptionMissingDataDirective, ExceptionNoSymbolValueFound {
002     if (eventHandler != null) {
003         eventHandler.BuildBinDirectivePre(step, this, line);
004     }
005
006     if (!line.isEmpty && line.isLineDirective && !
line.isLineOpCode) {
007         boolean isDirDchw = false;
008         boolean isDirDcb = false;
009         boolean isDirFlpDchw = false;
010         boolean isDirDcw0 = false;
011         boolean isDirDcw1 = false;
012         Object[] objs = null;
013
014         for (Token token : line.payload) {
015             lastToken = token;
016             ...
017         }
018     }
019 }
```

Part 1, of the AssemblerThumb class' BuildBinDirective method listing.

The `BuildBinDirective` method takes an integer and a `TokenLine` as arguments. On line 6 of the method, we check to see if the current token line has the correct attributes. If the line isn't empty, isn't an opcode, and is marked as a directive then we proceed to instantiate a number of method variables on lines 7 – 12. Each Boolean flag in this snippet represents a type of data directive. The variable, `objs`, is an array of object instances used to handle return data from the `ProcessSymbolValue` and `ProcessShift` support methods. Recall that we reviewed these methods during the support method section of the `AssemblyThumb` class review.

Recall from our review of the GenAsm assembler's supported directives, earlier in the text, that there are a number of data directives that are used to set the value of a certain address in the assembly program. On line 14 we iterate over the token line's payload and update the class field, `lastToken`, on line 15. We'll pick things up in the next section, and keep in mind that we'll be looking at code inside the main for loop.

*Listing 19-5. AssemblerThumb.java Main Method Details 5*

```
016 if (token.isDirective) {  
017     if (token.source.equals(JsonObjIsDirectives.NAME_DCHW)) {  
018         isDirDchw = true;  
019         isDirDcb = false;  
020         isDirFlpDchw = false;  
021         isDirDcw0 = false;  
022         isDirDcw1 = false;  
023     } else if (token.source.equals(JsonObjIsDirectives.NAME_DCW)) {  
024         isDirDcb = true;  
025         isDirDchw = false;  
026         isDirFlpDchw = false;  
027         isDirDcw0 = false;  
028         isDirDcw1 = false;  
029     } else if (token.source.equals(JsonObjIsDirectives.NAME_FLPDCHW)) {  
030         isDirDchw = true;  
031         isDirFlpDchw = true;  
032         isDirDcb = false;  
033         isDirDcw0 = false;  
034         isDirDcw1 = false;  
035     } else if (token.source.equals(JsonObjIsDirectives.NAME_DCWBF)) {
```

```

036     isDirDchw = true;
037     isDirDcw0 = true;
038     isDirFlpDchw = false;
039     isDirDcb = false;
040     isDirDcw1 = false;
041 } else if (token.source.equals(JsonObjIsDirectives.NAME_DCWBS)) {
042     isDirDchw = true;
043     isDirDcw1 = true;
044     isDirFlpDchw = false;
045     isDirDcb = false;
046     isDirDcw0 = false;
047 }
048 } else if (token.isDirectiveArg && (isDirDchw || isDirDcb)) {
049     if (token.type_name.equals(JsonObjIsEntryTypes.NAME_NUMBER) ==
true) { ...

```

Part 2, of the AssemblerThumb class' BuildBinDirective method listing.

Continuing our review of the `BuildBinDirective` method we take a look at the first snippet of code from inside the main for loop. Because this method is responsible for looking at each data directive encountered and creating a binary representation of that line of assembly source code. We check to see if the current token is a directive on line 16 and if so, we begin to process the different supported directives. The block of code that runs from line 17 to line 47 is responsible for detecting the directive type and setting the method variables, Boolean flags, to indicate the type of directive found.

On line 48 we perform another check on the current token. If the current token isn't a directive token, then we check to see if we've encountered a token argument, line 49. At this point in the code, line 50, we're going to check the different directive argument types starting with the number argument. In the next section, part 3, we'll see how we handle a directive's number argument.

#### *Listing 19-6. AssemblerThumb.java Main Method Details 6*

```

050 String resTmp;
051 Integer tInt = Utils.ParseNumberString(token.source);
052 resTmp = Integer.toBinaryString(tInt);

```

```

053 resTmp = Utils.FormatBinString(resTmp, lineBitSeries.bit_len);
054 tInt = Integer.parseInt(resTmp, 2);
055
056 if (lineNumRange != null) {
057     if (tInt < lineNumRange.min_value || tInt > lineNumRange.max_value)
058     {
059         throw new ExceptionNumberOutOfRange("Integer value " + tInt + " is outside of the specified range " + lineNumRange.min_value + " to " + lineNumRange.max_value + " for source '" + token.source + "' with line number " + token.lineNumAbs);
060     }
061     throw new ExceptionNoNumberRangeFound("Could not find number range for source '" + token.source + "' with line number " + token.lineNumAbs);
062 }
063
064 if (isDirFlpDchw == true) {
065     resTmp = UtilsEndianFlipBin(resTmp);
066 }
067
068 token.value = tInt;
069 if (isDirDchw == true) {
070     if (isDirDcw0 == true || isDirDcw1 == true) {
071         byte[] b = Utils.ToBytes(tInt);
072         short[] shorts = new short[2];
073         ByteBuffer.wrap(b).asShortBuffer().get(shorts);
074         byte bt;
075
076         if (isDirDcw0 == true) {
077             bt = b[b.length - 1];
078         } else {
079             bt = b[b.length - 2];
080         }
081
082         if ((int) bt < 0) {
083             tInt = bt & 0xFF;
084         } else {
085             tInt = (int) bt;
086         }
087
088         token.value = tInt;
089         resTmp = Integer.toBinaryString(tInt);
090         resTmp = Utils.FormatBinString(resTmp, lineBitSeries.bit_len);

```

```

091     line.payloadBinRepStrEndianBig1 = resTmp;
092     line.payloadBinRepStrEndianLil1 = UtilsEndianFlipBin(resTmp);
093
094 } else {
095     line.payloadBinRepStrEndianBig1 = resTmp;
096     line.payloadBinRepStrEndianLil1 = UtilsEndianFlipBin(resTmp);
097 }
098 } else if (isDirDcb == true) {
099     line.payloadBinRepStrEndianBig1 = resTmp;
100    line.payloadBinRepStrEndianLil1 = UtilsEndianFlipBin(resTmp);
101
102 } else {
103     throw new ExceptionMissingDataDirective("Could not find supported
data directive '" + token.source + "' with line number " +
token.lineNumAbs);
104 }

```

Part 3, of the AssemblerThumb class' BuildBinDirective method listing.

Part 3 of this method review, shown previously, is responsible for processing of a number directive argument. Local variables `resTmp` and `tInt` are defined on lines 50 – 51 and are used to parse the token's string source code and convert it to an `Integer` value, lines 52 – 54. Next up, on lines 56 – 62, we check to see if the number value is within the valid range. Recall that the ARM Thumb-1 instruction set has a 16-bit instruction set width. Thus, any numeric values defined using data directives have to be within the range 0 –  $2^{16}$ . If the numeric value is out of range, we throw an exception and exit the method.

The code on lines 64 – 66 is there to handle the case where we have a flipped data directive. These are directives that automatically convert their numeric data's binary encoding from big-endian to little-endian or vice versa. The token's value field is updated with the parsed integer value on line 68. If the Boolean flag `isDirDchw` is set to true, the code on lines 70 – 97 executes. The bulk of this code is associated with the if statement on lines 70 – 94. Let's take a look.

If the current directive token has its `isDirDcw0` or `isDirDcw1` data directive flag set to true, then we proceed to prepare a few local variables on

lines 71 – 74. The first entry, `b`, is an array of bytes that is generated by a call to the `ToBytes` method of the `Utils` class. The next variable is an array of shorts, line 72, that is populated with a call to the `ByteBuffer` class' `wrap` method. What this code does is convert a 2-element array of bytes into a single element array of shorts.

Lastly, on line 74 the temporary variable `bt` is declared. Because this block of code handles the `@DCWBF` and `@DCWBS` data directives. If you recall from our review of the assembler's data directives these particular directives allow you to target the first or second byte of a short. On lines 76 – 80 we check to see if the first or second byte is needed and update the `bt` local variable accordingly. Next, on lines 82 to 86 care is taken to force the byte values to be unsigned.

If the determined byte value is negative, line 82, then the byte value is added to the hex value `0xFF` using binary addition to force the byte to be unsigned. The `value` field of the current token object is updated on line 88 and the token line's big- and little-endian binary representation fields are updated on lines 89 – 92. The case where the data directive is not a single byte directive the code on lines 95 – 96 executes and updates the token line's big- and little-endian binary representation fields.

The case of the `@DCB` data directive is handled on lines 98 – 102 and the last few lines of the code block show the use of the missing data directive exception to exit the method in the case of an error. We'll pick things up in the next listing with the handling of labels as part of processing the directive token line.

#### *Listing 19-7. AssemblerThumb.java Main Method Details 7*

```
105 } else if (token.type_name.equals(JsonObjIsEntryTypes.NAME_LABEL_REF)
== true) {
106     String resTmp = "";
107     JsonObjIsDirectiveArg entry =
line.matchesDirective.get(0).args.get(0);
108     objs = ProcessSymbolValue(token.source, entry.bit_series,
token.lineNumAbs, line);
109     Integer tInt = (Integer) objs[0];
110     String label = (String) objs[1];
111     line.source.source += ";" + tInt + "]";
```

```

112
113     objs = ProcessShift(tInt, entry.num_range, entry.bit_shift,
entry.bit_series, token.source, token.lineNumAbs, true, (isDirDcw0 ==
false && isDirDcw1 == false));
114     resTmp = (String) objs[0];
115     tInt = (Integer) objs[1];
116
117     Logger.wrl("Symbol lookup: Found final symbol value: '" + tInt + "'"
for symbol, '" + label + "' at line " + line.lineNumAbs);
118     line.source.source += " (" +
Utils.Bin2Hex(Integer.toBinaryString(tInt)) + ") {" + tInt + "}";
119
120     if (isDirDcb == true && tInt > Byte.MAX_VALUE) {
121         throw new ExceptionMissingDataDirective("Could not find
supported data directive, cannot use a LabelRef that is greater than 255
bytes away, '" + token.source + "' with line number " + token.lineNumAbs);
122     }
123
124     token.value = tInt;
125     if (isDirFlpDchw == true) {
126         resTmp = UtilsEndianFlipBin(resTmp);
127     }
128
129     if (isDirDchw == true) {
130         if (isDirDcw0 == true || isDirDcw1 == true) {
131             byte[] b = Utils.ToBytes(tInt);
132             short[] shorts = new short[2];
133             ByteBuffer.wrap(b).asShortBuffer().get(shorts);
134
135             if (isDirDcw0 == true) {
136                 tInt = (int) shorts[0];
137             } else {
138                 tInt = (int) shorts[1];
139             }
140
141         if (entry.num_range != null) {
142             if (tInt < entry.num_range.min_value || tInt >
entry.num_range.max_value) {
143                 throw new ExceptionNumberOutOfRange("Integer value " +
tInt + " is outside of the specified range " + entry.num_range.min_value +
" to " + entry.num_range.max_value + " for source '" + token.source + "'"
with line number " + token.lineNumAbs);
144         }

```

```

145      }
146
147      token.value = tInt;
148      resTmp = Integer.toBinaryString(tInt);
149      resTmp = Utils.FormatBinString(resTmp,
150          lineBitSeries.bit_len);
151      line.payloadBinRepStrEndianBig1 = resTmp;
152      line.payloadBinRepStrEndianLill1 =
153          UtilsEndianFlipBin(resTmp);
154
155      } else {
156          resTmp = Utils.FormatBinString(resTmp,
157              entry.bit_series.bit_len);
158          line.payloadBinRepStrEndianBig1 = resTmp;
159          line.payloadBinRepStrEndianLill1 =
160              UtilsEndianFlipBin(resTmp);
161
162      }
163  } else {
164      throw new ExceptionMissingDataDirective("Could not find
165 supported data directive '" + token.source + "' with line number " +
166 token.lineNumAbs);
167
168 } else {
169     throw new ExceptionDirectiveArgNotSupported("Could not find
170 supported data directive '" + token.source + "' with line number " +
171 token.lineNumAbs);
172
173 }

```

Part 4, of the AssemblerThumb class' BuildBinDirective method listing.

Part 4 of the BuildBinDirectives method, shown previously, is responsible for processing a label reference argument. The block of code starts off on line 105 with a check to see if we're processing a label reference. If so, we proceed and declare a number of local variables for the task at hand,

lines 106 – 110. The first variable, `resTmp`, is a temporary variable used to hold formatted binary numbers, in string form. The next variable, `entry`, is used to hold a reference to the matching directive argument for the directive we're currently processing.

Subsequently, on line 108, a call to the support method `ProcessSymbolValue` is made. Feel free to look back to Chapter 17 to refresh your memory with regard to this method. The method is responsible for parsing the label reference and returning the correct value based on the label type, label reference character. Recall from previous chapters that we support four different ways of pulling data out of a label. On lines 109 – 110 we handle the method results, setting the returned value and the label name.

Note that on line 111 we append debugging information to the assembly source code so that we'll have some extra information to help with any issues concerning the label. By adjusting the assembly line's source code, appending a comment with the label's calculated value, we can ensure the added information shows up in the listing files generated by the assembly process. We're not done with the symbol's value just yet. We have to apply any value adjustments that are defined by the directive's argument definition.

This is handled on line 113 with a call to the `ProcessShift` support method. The results of the method call are stored on lines 114 and 115. The value stored in the `resTmp` local variable is a binary representation of the integer value stored in the `tInt` variable before its value is updated on line 115. Next, on line 118 we append two values to the original line of source code. We include a hexadecimal and integer representation of the shifted integer value. Again, this is done for debugging purposes. It really helps to be able to see what the symbol processor is doing and pick up on any issues with bit shifting etc.

Next up, on lines 120 – 122 we check to see if the determined value is within range, if not, we exit by throwing an exception. The token's value is updated on line 124 and we handle flipping the value on lines 125 – 127. If we're working on a flipped data directive. The last block of code in the listing runs from lines 129 – 165. In this code we handle the case of single byte data directives. On line 129 we check to see if we're processing a single byte data directive and on line 130, we test if we're working with the single byte 0 or single byte 1.

If so, the code on lines 131 – 151 executes. The code is responsible for pulling the target byte from the short value, lines 131 – 139, and checking to see if the number is valid, lines 141 – 145. If the value is within the valid range, we update the token and line fields as shown on lines 147 – 151. In the case where we’re processing a normal 2-byte data directive the code on lines 154 – 156 executes. In the case that we’re dealing with a single byte data directive, @DCB, similar code on lines 159 – 161 executes and updates the TokenLine’s fields as shown.

That brings us to the conclusion of the `BuildBinDirectives` method review. In the next section we’ll take a look at the first listing of the `BuildBinOpCodes` method. This method is on the other side of the coin with regard to the `BuildBinDirectives` method we’ve just looked at. I should mention that this method relies heavily on the `BuildBinOpCodePrep` method we reviewed in Chapter 17 along with the class’ other support methods. Due to the size of this method, I’ll list a skeleton of the method here first. We’ll then review the code blocks that are represented by comments in the listing one at a time.

*Listing 19-8. AssemblerThumb.java Main Method Details 8*

```
001 public void BuildBinOpCode(int step, TokenLine line) throws
ExceptionOpCodeAsArgument, ExceptionNoSymbolFound,
ExceptionUnexpectedTokenWithSubArguments, ExceptionNumberInvalidShift,
ExceptionNumberOutOfRange, ExceptionNoNumberRangeFound,
ExceptionUnexpectedTokenType, ExceptionInvalidEntry,
ExceptionInvalidAssemblyLine, ExceptionNoSymbolValueFound {
002     if (eventHandler != null) {
003         eventHandler.BuildBinOpCodePre(step, this, line);
004     }
005
006     if (!line.isEmpty && !line.isLineDirective &&
line.isLineOpCode) {
007         //Process BuildOpCodeEntry list creating a binary string
representation for each entry
008         BuildBinOpCodePrep(step, line);
009         List<BuildOpCodeThumb> buildEntries = line.buildEntries;
010         String res1 = "";
011         String resTmp1 = "";
012         String res2 = "";
```

```

013     String resTmp2 = "";
014     boolean inList = false;
015     boolean inGroup = false;
016     boolean inBxHi = false;
017     boolean inBxLo = false;
018     boolean inBl = false;
019     int[] inListRegisters = null;
020     BuildOpCodeThumb inListEntry = null;
021     BuildOpCodeThumb inGroupEntry = null;
022     BuildOpCodeThumb opCodeEntry = null;
023     Object[] objs = null;
024
025     for (BuildOpCodeThumb entry : buildEntries) {
026         resTmp1 = "";
027         resTmp2 = null;
028         if (entry.isOpCode) {
029             // <editor-fold desc="OpCode">
030             //...
031             // </editor-fold>
032         } else if (entry.isOpCodeArgList) {
033             // <editor-fold desc="OpCode Arg List">
034             //...
035             // </editor-fold>
036         } else if (entry.isOpCodeArgGroup) {
037             // <editor-fold desc="OpCode Arg Group">
038             //...
039             // </editor-fold>
040         } else if (entry.isOpCodeArg) {
041             // <editor-fold desc="OpCode Arg">
042             //...
043             // </editor-fold>
044         }
045         entry.binRepStr1 = resTmp1;
046         res1 += resTmp1;
047
048         if (Utils.IsEmpty(resTmp2) == false) {
049             entry.binRepStr2 = resTmp2;
050             res2 += resTmp2;
051         } else {
052             entry.binRepStr2 = null;
053         }
054     }

```

```

325      //Clean out non-bit series entries from the build entries list
and sort by bit series desc
326      BuildOpCodeEntryThumbSorter buildEntriesSorter = new
BuildOpCodeEntryThumbSorter(BuildOpCodeEntryThumbSorterType.BIT_SERIES_DSC
);
327      buildEntriesSorter.Clean(buildEntries);
328      Collections.sort(buildEntries, buildEntriesSorter);
329
330      res1 = "";
331      res2 = "";
332      for (BuildOpCodeThumb entry : buildEntries) {
333          res1 += entry.binRepStr1;
334          if (Utils.IsEmpty(entry.binRepStr2) == false) {
335              res2 += entry.binRepStr2;
336          }
337      }
338      line.payloadBinRepStrEndianBig1 = res1;
339      line.payloadBinRepStrEndianLil1 = UtilsEndianFlipBin(res1);
340
341      if (Utils.IsEmpty(res2) == false) {
342          line.payloadBinRepStrEndianBig2 = res2;
343          line.payloadBinRepStrEndianLil2 = UtilsEndianFlipBin(res2);
344      }
345  } else if (line.isLineLabelDef || line.isLineDirective &&
line.isLineOpCode) {
346      throw new ExceptionInvalidAssemblyLine("Could not find a valid
assembly line entry for the given AREA with OpCode line source '" +
line.source.source + "' and line number " + line.lineNumAbs + ", " +
line.isEmpty + ", " + line.isLineDirective + ", " +
line.isLineOpCode);
347  }
348
349  if (eventHandler != null) {
350      eventHandler.BuildBinOpCodePost(step, this);
351  }
352 }

```

A skeleton of the AssemblerThumb class' BuildBinOpCode method. This listing shows the main method code with certain code blocks represented by comments with the actual code removed.

Recall that the `BuildBinOpCode` method is called after the `BuildBinDirective` method for active lines of code. We start the method off on line 6 with a check to see if the `TokenLine` has the proper attributes. Namely we're looking for non-empty opcode lines to process. A very important support method call is made on line 8. We covered this method in Chapter 17 during the support method review. It's responsible for prepping each token line for use by the `BuildBinOpCode` method.

Local method variables are declared on lines 9 – 23. The first entry is a List of `BuildOpCodeThumb` objects, `buildEntries`, and it's set to reference the current line's `buildEntries` field. This field is initialized by the call to the `BuildBinOpCodePrep` method. The next four variables on lines 10 – 13 are temporary variables used in creating the binary representation for the current token line, and in some cases with BL opcodes, a second line of binary representation is needed. The next block of local variables, lines 14 – 18, are used to identify if we've encountered some of the different `BuildOpCodeThumb` entry types.

I should mention that we're going to be iterating over `buildEntries` as opposed to `Tokens`. The `BuildOpCodeThumb` class actually connects `Tokens` to the defined arguments for a given opcode so iterating over a list of them is not too different from iterating over a list of `Tokens`. The next variable listed, `inListRegisters`, line 19, is used to track which registers are being used in a given list. Register ranges are broken into individual registers and the value of that register in the `inListRegisters` array is set to 1. Unused registers are set to 0.

The next three variables are used to track important `BuildOpCodeThumb` objects that indicate the start of a list or group opcode argument. Lastly, the `obj` variable is used to hold results from certain support methods used to process symbols during the binary encoding process. The main loop begins on line 25 where we iterate over the `BuildOpCodeEntries` on a given `TokenLine`. The local variables used to aggregate the binary encoding from each entry, thus creating an encoding for the current line, are reset on lines 26 – 27.

This core work done by this method occurs on lines 28 – 313. This code block has been converted into a skeleton for brevity. Using this

approach, we can clearly see the categories being handled while abstracting how they are handled. All we need to know at this point in time is that at the end of this block of code the `resTmp1`, and possibly `resTmp2`, variables hold the binary encoding for this line. Fear not, we'll review these code blocks in detail in just a bit. Let's take a look at the different entries that are handled by this large block of code.

On line 28 we detect and handle an opcode entry, lines 29 – 40, followed by the if-else statement on line 41 which is used to determine if we're processing a list opcode argument, lines 42 – 145. Similarly, the code on lines 146 – 210 is responsible for processing group opcode arguments. Lastly, the bit of code on lines 211 – 312 is responsible for handling all remaining opcode arguments. At the end of this code block, 28 – 313, we have the binary representation of this entry stored in the `resTmp1` variable and as such we update the entry's `binRepStr1` field on line 314 and aggregate the binary encoding for the current token line on line 315.

We mentioned that in some cases we'll have two lines of binary representation for the current token line. In such a case the code on lines 317 – 320 executes and the entry's `binRepStr2` field is updated, in the same way `binRepStr1` was, on line 318 followed by an adjustment to the `res2` variable. The remaining code on lines 326 – 344 is redundant but it's an important safety check. We want to make sure everything that we're planning to put in the final binary representation of this program should actually be there.

To that end we rebuild the token line's binary representation after we sort and clean the list of build entries. Finally, on lines 345 – 347, if we're not working with the type of token line we expect, then we throw an exception and exit the method. That brings us to the conclusion of the `BuildBinOpCode` method's skeleton review. We'll pick things back up by looking at how this method handles an opcode entry, lines 29 – 40, in the next listing.

*[Listing 19-9. AssemblerThumb.java Main Method Details 9](#)*

```
029 // <editor-fold desc="OpCode">
030 opCodeEntry = entry;
031 resTmp1 = entry.opCode.bit_rep.bit_string;
032
033 if (entry.opCode.index == JsonObjIsOpCodes.BX_HI_INDEX) {
```

```

034     inBxHi = true;
035 } else if (entry.opCode.index == JsonObjIsOpCodes.BX_LO_INDEX) {
036     inBxLo = true;
037 } else if (entry.opCode.index == JsonObjIsOpCodes.BL_INDEX) {
038     inBl = true;
039 }
040 // </editor-fold>

```

A listing showing the handling of an opcode entry in the BuildBinOpCode method.

The snippet of code shown in the previous listing is used to handle encountering an opcode entry while iterating over the BuildOpCodeThumb entries, buildEntries, for a given TokenLine. The code is fairly simple. We track the opcode by updating the opCodeEntry local variable on line 30 and we set the resTmp1 variable with the known binary string representing the current opcode on line 31. Remember, the resTmp1 and resTmp2 variables are meant to hold the binary representation of the current entry and are aggregated to create the binary representation for the entire token line. In the following listing we'll take a look at the code block that handles the list opcode argument, lines 42 – 145.

#### *Listing 19-10. AssemblerThumb.java Main Method Details 10*

```

042 // <editor-fold desc="OpCode Arg List">
043 if
(entry.tokenOpCodeArgList.type_name.equals(JsonObjIsEntryTypes.NAME_REGISTER_LOW)) {
044     if (entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_0))
{
045         inListRegisters[0] = 1;
046
047     } else if
(entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_1)) {
048         inListRegisters[1] = 1;
049
050     } else if
(entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_2)) {
051         inListRegisters[2] = 1;
052

```

```
053     } else if
(entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_3)) {
054         inListRegisters[3] = 1;
055
056     } else if
(entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_4)) {
057         inListRegisters[4] = 1;
058
059     } else if
(entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_5)) {
060         inListRegisters[5] = 1;
061
062     } else if
(entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_6)) {
063         inListRegisters[6] = 1;
064
065     } else if
(entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_7)) {
066         inListRegisters[7] = 1;
067
068     } else {
069         throw new ExceptionInvalidEntry("Found invalid LOW register
entry '" + entry.tokenOpCodeArgList.source + "' for line source '" +
line.source.source + " and line number " + line.lineNumAbs);
070     }
071 } else if
(entry.tokenOpCodeArgList.type_name.equals(JsonObjIsEntryTypes.NAME_REGIST
ER_HI)) {
072     if (entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_8))
{
073         inListRegisters[0] = 1;
074
075     } else if
(entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_9)) {
076         inListRegisters[1] = 1;
077
078     } else if
(entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_10)) {
079         inListRegisters[2] = 1;
080
081     } else if
(entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_11)) {
082         inListRegisters[3] = 1;
```

```

083
084     } else if
(entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_12)) {
085         inListRegisters[4] = 1;
086
087     } else if
(entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_13) ||
entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_SP)) {
088         inListRegisters[5] = 1;
089
090     } else if
(entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_14) ||
entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_LR)) {
091         inListRegisters[6] = 1;
092
093     } else if
(entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_15) ||
entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_PC)) {
094         inListRegisters[7] = 1;
095
096     } else {
097         throw new ExceptionInvalidEntry("Found invalid HI register entry
'" + entry.tokenOpCodeArgList.source + "' for line source '" +
line.source.source + " and line number " + line.lineNumAbs);
098     }
099 } else if
(entry.tokenOpCodeArgList.type_name.equals(JsonObjIsEntryTypes.NAME_REGISTER_PC)) {
100     if (entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_15)
|| entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_LR)) {
101         inListRegisters[7] = 1;
102     } else {
103         throw new ExceptionInvalidEntry("Found invalid PC register entry
'" + entry.tokenOpCodeArgList.source + "' for line source '" +
line.source.source + " and line number " + line.lineNumAbs);
104     }
105 } else if
(entry.tokenOpCodeArgList.type_name.equals(JsonObjIsEntryTypes.NAME_REGISTER_SP)) {
106     if (entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_13)
|| entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_SP)) {
107         inListRegisters[5] = 1;
108     } else {

```

```

109      throw new ExceptionInvalidEntry("Found invalid SP register entry
'" + entry.tokenOpCodeArgList.source + "' for line source '" +
line.source.source + " and line number " + line.lineNumAbs);
110  }
111 } else if
(entry.tokenOpCodeArgList.type_name.equals(JsonObjIsEntryTypes.NAME_REGISTER_LR)) {
112   if (entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_14)
|| entry.tokenOpCodeArgList.source.equals(JsonObjIsRegisters.R_LR)) {
113     inListRegisters[6] = 1;
114   } else {
115     throw new ExceptionInvalidEntry("Found invalid LR register entry
'" + entry.tokenOpCodeArgList.source + "' for line source '" +
line.source.source + " and line number " + line.lineNumAbs);
116   }
117 } else if
(entry.tokenOpCodeArgList.type_name.equals(JsonObjIsEntryTypes.NAME_LABEL_REF)) {
118   throw new ExceptionInvalidEntry("Found invalid LABEL entry '" +
entry.tokenOpCodeArgList.source + "' for line source '" +
line.source.source + " and line number " + line.lineNumAbs);
119
120 } else if
(entry.tokenOpCodeArgList.type_name.equals(JsonObjIsEntryTypes.NAME_REGISTERWB)) {
121   throw new ExceptionInvalidEntry("Found invalid REGISTERWB entry '" +
entry.tokenOpCodeArgList.source + "' for line source '" +
line.source.source + " and line number " + line.lineNumAbs);
122
123 } else if
(entry.tokenOpCodeArgList.type_name.equals(JsonObjIsEntryTypes.NAME_NUMBER)) {
124   throw new ExceptionInvalidEntry("Found invalid NUMBER entry '" +
entry.tokenOpCodeArgList.source + "' for line source '" +
line.source.source + " and line number " + line.lineNumAbs);
125
126 } else if
(entry.tokenOpCodeArgList.type_name.equals(JsonObjIsEntryTypes.NAME_STOP_LIST)) {
127   inList = false;
128   String ts = "";
129   for (int z = 0; z < inListRegisters.length; z++) {
130     if (inListRegisters[z] == 1) {

```

```

131         ts = ("1" + ts);
132     } else {
133         ts = ("0" + ts);
134     }
135 }
136 inListEntry.binRepStr1 = ts;
137 inListRegisters = null;
138
139 } else if
(entry.tokenOpCodeArgList.type_name.equals(JsonObjIsEntryTypes.NAME_STOP_G
ROUP)) {
140     throw new ExceptionInvalidEntry("Found invalid STOP GROUP entry ''"
+ entry.tokenOpCodeArgList.source + "' for line source '" +
line.source.source + " and line number " + line.lineNumAbs);
141
142 } else {
143     throw new ExceptionUnexpectedTokenType("Found unexpected LIST sub-
token type '" + entry.tokenOpCodeArgList.type_name + "' for line source ''"
+ line.source.source + " and line number " + line.lineNumAbs);
144 }
145 //</editor-fold>
```

A listing showing the handling of an opcode list argument entry in the BuildBinOpCode method.

The previous listing shows the missing code block from the skeleton review of the BuildBinOpCode method. It's a rather lengthy code block but it doesn't have much in the way of complexity. Look over the method and note that its only purpose is to process the registers used by the list argument. Where is the final binary representation created you ask? Can you see it in the list of entry types handled by the code? If you thought, NAME\_STOP\_LIST, then you thought right. All the cases handled by the code are specifically for toggling a register in the array of available registers.

We don't have to worry about register ranges because we've unrolled those into individual register entries in a previous step. When the list stop entry is encountered, we scan through the array of registers and create a binary representation of them on lines 127 – 137. There is a little magic going

on here. Notice that on lines 136 and 137 when we finalize the binary representation of the register list that we don't set the `resTmp1` variable.

How can this be? Won't it set the binary representation of all the entries to a blank string? Yes, it will. Then how does the list get a proper binary representation set? Well, notice that the binary representation is created when the list stop entry is encountered. This happens after the list start entry, and it updates the list start entry. In short because the last entry in the list updates the first entry in the list, the binary representation gets carried through. A caveat of this implementation is that list entries, other than the starting entry, do not have their individual binary encodings set.

*Listing 19-11. AssemblerThumb.java Main Method Details 11*

```
147 // <editor-fold desc="OpCode Arg Group">
148 if
(entry.tokenOpCodeArgGroup.type_name.equals(JsonObjIsEntryTypes.NAME_REGIS
TER_LOW)) {
149     resTmp1 = entry.tokenOpCodeArgGroup.register.bit_rep.bit_string;
150
151 } else if
(entry.tokenOpCodeArgGroup.type_name.equals(JsonObjIsEntryTypes.NAME_REGIS
TER_HI)) {
152     resTmp1 = entry.tokenOpCodeArgGroup.register.bit_rep.bit_string;
153
154 } else if
(entry.tokenOpCodeArgGroup.type_name.equals(JsonObjIsEntryTypes.NAME_REGIS
TER_PC)) {
155     resTmp1 = entry.tokenOpCodeArgGroup.register.bit_rep.bit_string;
156
157 } else if
(entry.tokenOpCodeArgGroup.type_name.equals(JsonObjIsEntryTypes.NAME_REGIS
TER_SP)) {
158     resTmp1 = entry.tokenOpCodeArgGroup.register.bit_rep.bit_string;
159
160 } else if
(entry.tokenOpCodeArgGroup.type_name.equals(JsonObjIsEntryTypes.NAME_REGIS
TER_LR)) {
161     resTmp1 = entry.tokenOpCodeArgGroup.register.bit_rep.bit_string;
162 }
```

```

163 } else if
(entry.tokenOpCodeArgGroup.type_name.equals(JsonObjIsEntryTypes.NAME_LABEL
_REF)) {
164     objs = ProcessSymbolValue(entry.tokenOpCodeArgGroup.source,
entry.opCodeArgGroup.bit_series, entry.tokenOpCodeArgGroup.lineNumAbs,
line);
165     Integer tInt = (Integer) objs[0];
166     String label = (String) objs[1];
167     line.source.source += ";" + [ " + tInt + "];
168
169     //special rule for ADD OpCode
170     if (opCodeEntry.binRepStr1.equals(SPECIAL_ADD_OP_CODE_CHECK) &&
tInt < 0) {
171         opCodeEntry.binRepStr1 =
JsonObjIsOpCodes.ADD_OP_CODE_SPECIAL; //101100001;
172         tInt *= -1;
173     }
174
175     objs = ProcessShift(tInt, entry.opCodeArgGroup.num_range,
entry.opCodeArgGroup.bit_shift, entry.opCodeArgGroup.bit_series,
entry.tokenOpCodeArgGroup.source, entry.tokenOpCodeArgGroup.lineNumAbs,
false, true);
176     resTmp1 = (String) objs[0];
177     tInt = (Integer) objs[1];
178
179     Logger.wrl("Symbol lookup: Found final symbol value: '" + tInt + "'"
for symbol, '' + label + "' at line " + line.lineNumAbs);
180     line.source.source += "(" +
Utils.Bin2Hex(Integer.toBinaryString(tInt)) + ") {" + tInt + "}";
181     entry.tokenOpCodeArgGroup.value = tInt;
182
183 } else if
(entry.tokenOpCodeArgGroup.type_name.equals(JsonObjIsEntryTypes.NAME_REGIS
TERWB)) {
184     resTmp1 = entry.tokenOpCodeArgGroup.register.bit_rep.bit_string;
185
186 } else if
(entry.tokenOpCodeArgGroup.type_name.equals(JsonObjIsEntryTypes.NAME_NUMBE
R)) {
187     Integer tInt =
Utils.ParseNumberString(entry.tokenOpCodeArgGroup.source);
188
189     //special rule for ADD OpCode

```

```

190     if (opCodeEntry.binRepStr1.equals(SPECIAL_ADD_OP_CODE_CHECK) &&
tInt < 0) {
191         opCodeEntry.binRepStr1 =
JsonObjIsOpCodes.ADD_OP_CODE_SPECIAL; // "101100001";
192         tInt *= -1;
193     }
194
195     objs = ProcessShift(tInt, entry.opCodeArgGroup.num_range,
entry.opCodeArgGroup.bit_shift, entry.opCodeArgGroup.bit_series,
entry.tokenOpCodeArgGroup.source, entry.tokenOpCodeArgGroup.lineNumAbs,
false, true);
196     resTmp1 = (String) objs[0];
197     tInt = (Integer) objs[1];
198     entry.tokenOpCodeArgGroup.value = tInt;
199
200 } else if
(entry.tokenOpCodeArgGroup.type_name.equals(JsonObjIsEntryTypes.NAME_STOP_
LIST)) {
201     throw new ExceptionInvalidEntry("Found invalid STOP LIST entry '" +
entry.tokenOpCodeArgList.source + "' for line source '" +
line.source.source + " and line number " + line.lineNumAbs);
202
203 } else if
(entry.tokenOpCodeArgGroup.type_name.equals(JsonObjIsEntryTypes.NAME_STOP_
GROUP)) {
204     inGroup = false;
205     inGroupEntry = null;
206
207 } else {
208     throw new ExceptionUnexpectedTokenType("Found unexpected GROUP sub-
token type '" + entry.tokenOpCodeArgGroup.type_name + "' for line source
'" + line.source.source + " and line number " + line.lineNumAbs);
209 }
210 //</editor-fold>
```

A listing showing the handling of an opcode group argument entry in the BuildBinOpCode method.

The code shown in the previous listing is responsible for handling the group opcode argument. This code runs from line 147 – 210 and fits into the

“OpCode Arg Group” comment in the `BuildBinOpCode` method’s skeleton outline. On lines 148 – 162 we handle and register entries that are a part of the group. In each case we simply copy over the binary encoding for the given register to the `resTmp1` local variable. A slightly more complex case, that’s handled by this code, occurs on lines 163 – 182 with the occurrence of a label reference entry. This requires a bit more work for us to do. We need to process the symbol value and apply any numeric shifts and adjustments that are defined by the opcode group argument.

On line 164 we call the support method, `ProcessSymbolValue`, with a number of arguments that describe the current context we’re working in. The results of the method call, a numeric value and a clean label name, are stored in local variables `tInt` and `label` on lines 165 – 166. In a similar way to how we handled directive label reference arguments we append the determined value to the assembly source code, line 167, so that it will end up in the listing output file. On lines 170 – 173 we handle the specific case of the ADD opcode that requires us to toggle one bit after we know the value expressed as an argument is positive or negative.

Next up, we process any shifts or other adjustments that need to be made to the numeric value before it’s used, line 175. The result from the support method call gives us an integer value and a binary representation of that integer value. This information is used to append more debugging info to the token line’s source code. This is similar to the symbol comments we’ve seen appended earlier on in this method review. Lastly, we store the calculated integer value with the opcode argument group token on line 181. Note that all we need to do to ensure this binary representation is handled properly is to set the `resTmp1` variable, line 176.

The only other snippet of code worth mentioning is the handling of the `NAME_NUMBER` entry type on lines 187 – 198. This code is very similar to the label reference code we looked at earlier. Take a moment to look over it and make sure you understand it before moving on. I should mention that on lines 204 and 205, when the stop group entry type is encountered, we set the `inGroup` variable to false indicating we’re no longer in a group opcode argument. That brings us to the conclusion of this code block review. We have one more to go, the block of code responsible for handling the remaining opcode argument types.

*Listing 19-12. AssemblerThumb.java Main Method Details 12*

```
212 // <editor-fold desc="OpCode Arg">
213 if
(entry.tokenOpCodeArg.type_name.equals(JsonObjIsEntryTypes.NAME_REGISTER_L
OW)) {
214     resTmp1 = entry.tokenOpCodeArg.register.bit_rep.bit_string;
215     if (inBxLo) {
216         resTmp1 += "000"; //empty register
217         inBxLo = false;
218     }
219 } else if
(entry.tokenOpCodeArg.type_name.equals(JsonObjIsEntryTypes.NAME_REGISTER_H
I)) {
220     resTmp1 = entry.tokenOpCodeArg.register.bit_rep.bit_string;
221     if (inBxHi) {
222         resTmp1 += "000"; //empty register
223         inBxHi = false;
224     }
225 } else if
(entry.tokenOpCodeArg.type_name.equals(JsonObjIsEntryTypes.NAME_REGISTER_P
C)) {
226     resTmp1 = entry.tokenOpCodeArg.register.bit_rep.bit_string;
227     if (inBxHi) {
228         resTmp1 += "000"; //empty register
229         inBxHi = false;
230     }
231 } else if
(entry.tokenOpCodeArg.type_name.equals(JsonObjIsEntryTypes.NAME_REGISTER_S
P)) {
232     resTmp1 = entry.tokenOpCodeArg.register.bit_rep.bit_string;
233     if (inBxHi) {
234         resTmp1 += "000"; //empty register
235         inBxHi = false;
236     }
237 } else if
(entry.tokenOpCodeArg.type_name.equals(JsonObjIsEntryTypes.NAME_REGISTER_L
R)) {
238     resTmp1 = entry.tokenOpCodeArg.register.bit_rep.bit_string;
239     if (inBxHi) {
240         resTmp1 += "000"; //empty register
241         inBxHi = false;
242 }
```

```

243 } else if
(entry.tokenOpCodeArg.type_name.equals(JsonObjIsEntryTypes.NAME_LABEL_REF)
) {
244     objs = ProcessSymbolValue(entry.tokenOpCodeArg.source,
entry.opCodeArg.bit_series, entry.tokenOpCodeArg.lineNumAbs, line);
245     Integer tInt = (Integer) objs[0];
246     String label = (String) objs[1];
247     line.source.source += " [" + tInt + "]";
248
249     //special rule for ADD OpCode
250     if (opCodeEntry.binRepStr1.equals(SPECIAL_ADD_OP_CODE_CHECK) &&
tInt < 0) {
251         opCodeEntry.binRepStr1 =
JsonObjIsOpCodes.ADD_OP_CODE_SPECIAL; // "101100001";
252         tInt *= -1;
253     }
254
255     objs = ProcessShift(tInt, entry.opCodeArg.num_range,
entry.opCodeArg.bit_shift, entry.opCodeArg.bit_series,
entry.tokenOpCodeArg.source, entry.tokenOpCodeArg.lineNumAbs, false,
true);
256     resTmp1 = (String) objs[0];
257     tInt = (Integer) objs[1];
258     String blResTmp1 = (String) objs[2];
259     Integer bltInt = (Integer) objs[3];
260
261     if (inBl == true) {
262         //in BL OpCode which generates 4 bytes of instructions instead
of two
263         if (bltInt < 0) {
264             resTmp1 = Utils.FormatBinString(blResTmp1, 23, true, "1");
265         } else {
266             resTmp1 = Utils.FormatBinString(blResTmp1, 23, true, "0");
267         }
268         String nResTmp = resTmp1.substring(0, resTmp1.length() - 1);
269         String halfHi = nResTmp.substring(0, 11);
270         String halfLo = nResTmp.substring(11);
271         resTmp1 = halfHi;
272         resTmp2 = jsonObjIsOpCodes.BL_OP_CODE_BIN_ENTRY_2 + halfLo;
273         tInt = Integer.parseInt(resTmp1, 2);
274         inBl = false;
275         line.byteLength = 4;
276     }

```

```

277
278     Logger.wrl("Symbol lookup: Found final symbol value: '" + tInt + "'"
for symbol, '" + label + "' at line " + line.lineNumAbs);
279     line.source.source += "(" +
Utils.Bin2Hex(Integer.toBinaryString(tInt)) + ")" {" + tInt + "}";
280     entry.tokenOpCodeArg.value = tInt;
281
282 } else if
(entry.tokenOpCodeArg.type_name.equals(JsonObjIsEntryTypes.NAME_REGISTERWB)
) {
283     resTmp1 = entry.tokenOpCodeArg.register.bit_rep.bit_string;
284
285 } else if
(entry.tokenOpCodeArg.type_name.equals(JsonObjIsEntryTypes.NAME_NUMBER)) {
286     Integer tInt =
Utils.ParseNumberString(entry.tokenOpCodeArg.source);
287
288     if (entry.opCodeArg.num_range.handle_prefetch) {
289         tInt -= pcPreFetchHalfwords;
290     }
291
292     //special rule for ADD OpCode '101100000'
293     if (opCodeEntry.binRepStr1.equals(SPECIAL_ADD_OP_CODE_CHECK) &&
tInt < 0) {
294         opCodeEntry.binRepStr1 =
JsonObjIsOpCodes.ADD_OP_CODE_SPECIAL; //"101100001";
295         tInt *= -1;
296     }
297
298     objs = ProcessShift(tInt, entry.opCodeArg.num_range,
entry.opCodeArg.bit_shift, entry.opCodeArg.bit_series,
entry.tokenOpCodeArg.source, entry.tokenOpCodeArg.lineNumAbs, false,
true);
299     resTmp1 = (String) objs[0];
300     tInt = (Integer) objs[1];
301     entry.tokenOpCodeArg.value = tInt;
302 } else if
(entry.tokenOpCodeArg.type_name.equals(JsonObjIsEntryTypes.NAME_START_LIST)
) {
303     inList = true;
304     inListRegisters = new int[8];
305     inListEntry = entry;

```

```

306 } else if
(entry.tokenOpCodeArg.type_name.equals(JsonObjIsEntryTypes.NAME_START_GROUP)) {
307     inGroup = true;
308     inGroupEntry = entry;
309 } else {
310     throw new ExceptionUnexpectedTokenType("Found unexpected token type
'" + entry.tokenOpCodeArg.type_name + "' for line source '" +
line.source.source + " and line number " + line.lineNumAbs);
311 }
312 // </editor-fold>

```

A listing showing the handling of an opcode argument entry in the `BuildBinOpCode` method.

The last block of code that we have to review from the `BuildBinOpCode` method skeleton is responsible for processing any basic opcode arguments. With regard to list and group opcode arguments only the starting entry is handled in this code block as we'll soon see. The code on lines 213 – 242 is responsible for handling certain registers that can be used as an opcode argument. In each case we set the `resTmp1` local variable to the binary encoding of the given register. We also check for specific opcodes that require us to add some padding values to the binary encoding. This is due to the defined structure of the BX opcode's arguments.

The next snippet of code in listing, lines 243 – 280, is responsible for handling a label reference opcode argument. This code is very similar to the label reference code we've seen previously so we'll only focus on the differences. The main difference resides on lines 261 – 276. This is a special case caused by the BL opcode that requires us to create a second line of binary code that will be injected into the final representation of the program directly after the current line.

Take a moment to review the code on lines 286 – 301 where a numeric value is handled. We've seen this code before but, take a moment to review it and understand it before moving on. The only other snippets of code worth mentioning are on lines 303 – 305 and 307 – 308. Notice that these lines of code are used to end a list or group opcode argument. That wraps up our

review of the class' main methods. In the next section we'll take a look at the `AssemblerThumb` class in action.

## Demonstration: AssemblerThumb.java

For the `AssemblerThumb` class demonstration I wanted us to look at the interim output files generated by the assembler. This is going to be a rather large demonstration section. What I'm hoping to do is illustrate the changes made to the assembly source code as it's processed. We'll use a very simple assembly source code program, listed here.

*Listing 19-13. AssemblerThumb.java Demonstration 1*

```
//genasm_source.txt
01 @TTL |AssemblerDemo|
02 @SUBT |SubTitleTest|
03 @AREA |Program| @CODE, @READONLY
04 @ENTRY
05
06     LDR      R0, [PC, #0]
07
08 @END
```

The assembly source code program that is used to demonstrate the `AssemblerThumb` class.

The previous listing shows the original source code for this demonstration. As you can see it's exceedingly simple. But it serves as a good demonstration, showing how the assembly text is lexerized, tokenized, and refactored throughout the assembly process.

*Listing 19-14. AssemblerThumb.java Demonstration 2*

```
//output_pre_processed_assembly.txt
01 LDR      R0, [PC, #0]
```

The output of the preprocessor, only active rows from the assembly source code file.

In the previous listing we are shown the active line output of the preprocessor. Notice that no changes have been made to the assembly source code. In the next listing we'll see the output generated by the lexer.

*Listing 19-15. AssemblerThumb.java Demonstration 3*

```
//output_lexer.json
001 {
002 "obj_name": "ArtifactLine",
003 "lineNum": 5,
004 "sourceLen": 23,
005 "source": "    LDR      R0, [PC, #0]",
006 "payload": [
007     {
008         "obj_name": "Artifact",
009         "posStart": 3,
010         "posStop": 5,
011         "len": 3,
012         "lineNum": 5,
013         "index": 0,
014         "payload": "LDR"
015     },
016     {
017         "obj_name": "Artifact",
018         "posStart": 11,
019         "posStop": 12,
020         "len": 2,
021         "lineNum": 5,
022         "index": 1,
023         "payload": "R0"
024     },
025     {
026         "obj_name": "Artifact",
027         "posStart": 15,
028         "posStop": 15,
029         "len": 1,
030         "lineNum": 5,
031         "index": 2,
032         "payload": "["
033     },
034     {
```

```

035     "obj_name": "Artifact",
036     "posStart": 16,
037     "posStop": 17,
038     "len": 2,
039     "lineNum": 5,
040     "index": 3,
041     "payload": "PC"
042   },
043   {
044     "obj_name": "Artifact",
045     "posStart": 20,
046     "posStop": 21,
047     "len": 2,
048     "lineNum": 5,
049     "index": 4,
050     "payload": "#0"
051   },
052   {
053     "obj_name": "Artifact",
054     "posStart": 22,
055     "posStop": 22,
056     "len": 1,
057     "lineNum": 5,
058     "index": 5,
059     "payload": "]"
060   }
061 ] }

```

The output of the lexer, all operative text has been converted to `ArtifactLine` and `Artifact` objects.

Notice that all non-white space pieces of text on a given line have been converted to `Artifact` objects that belong to the `ArtifactLine`'s payload. In the next listing we'll take a look at the data structure after the tokenizer processes it.

#### *Listing 19-16. AssemblerThumb.java Demonstration 4*

```
//output_tokened_phase0_tokenized.json
001 {
```

```
002 "obj_name": "TokenLine",
003 "byteLength": 2,
004 "addressInt": 0,
005 "lineNumActive": 0,
006 "lineNumAbs": 5,
007 "payloadLen": 6,
008 "payloadLenArg": 0,
009 "isLineEmpty": false,
010 "isLineOpCode": false,
011 "isLineDirective": false,
012 "isLineLabelDef": false,
013 "source": { ... },
014 "payload": [
015     {
016         "obj_name": "Token",
017         "type_name": "OpCode",
018         "source": "LDR",
019         "lineNumAbs": 5,
020         "index": 0,
021         "payloadLen": 0,
022         "payloadArgLen": 0,
023         "isOpCodeArg": false,
024         "isDirectiveArg": false,
025         "isOpCode": true,
026         "isDirective": false,
027         "isComment": false,
028         "isLabel": false,
029         "isLabelRef": false,
030         "artifact": { ... },
031         "type": { ... },
032         "payload": []
033     },
034     {
035         "obj_name": "Token",
036         "type_name": "RegisterLow",
037         "source": "R0",
038         "lineNumAbs": 5,
039         "index": 1,
040         "payloadLen": 0,
041         "payloadArgLen": 0,
042         "isOpCodeArg": false,
043         "isDirectiveArg": false,
044         "isOpCode": false,
```

```
045     "isDirective": false,
046     "isComment": false,
047     "isLabel": false,
048     "isLabelRef": false,
049     "artifact": { ... },
050     "type": { ... },
051     "payload": []
052   },
053   {
054     "obj_name": "Token",
055     "type_name": "GroupStart",
056     "source": "[",
057     "lineNumAbs": 5,
058     "index": 2,
059     "payloadLen": 0,
060     "payloadArgLen": 0,
061     "isOpCodeArg": false,
062     "isDirectiveArg": false,
063     "isOpCode": false,
064     "isDirective": false,
065     "isComment": false,
066     "isLabel": false,
067     "isLabelRef": false,
068     "artifact": { ... },
069     "type": { ... },
070     "payload": []
071   },
072   {
073     "obj_name": "Token",
074     "type_name": "RegisterPc",
075     "source": "PC",
076     "lineNumAbs": 5,
077     "index": 3,
078     "payloadLen": 0,
079     "payloadArgLen": 0,
080     "isOpCodeArg": false,
081     "isDirectiveArg": false,
082     "isOpCode": false,
083     "isDirective": false,
084     "isComment": false,
085     "isLabel": false,
086     "isLabelRef": false,
087     "artifact": { ... },
```

```
088     "type": { ... },
089     "payload": []
090   },
091   {
092     "obj_name": "Token",
093     "type_name": "Number",
094     "source": "#0",
095     "lineNumAbs": 5,
096     "index": 4,
097     "payloadLen": 0,
098     "payloadArgLen": 0,
099     "isOpCodeArg": false,
100     "isDirectiveArg": false,
101     "isOpCode": false,
102     "isDirective": false,
103     "isComment": false,
104     "isLabel": false,
105     "isLabelRef": false,
106     "artifact": { ... },
107     "type": { ... },
108     "payload": []
109   },
110   {
111     "obj_name": "Token",
112     "type_name": "GroupStop",
113     "source": "]",
114     "lineNumAbs": 5,
115     "index": 5,
116     "payloadLen": 0,
117     "payloadArgLen": 0,
118     "isOpCodeArg": false,
119     "isDirectiveArg": false,
120     "isOpCode": false,
121     "isDirective": false,
122     "isComment": false,
123     "isLabel": false,
124     "isLabelRef": false,
125     "artifact": { ... },
126     "type": { ... },
127     "payload": []
128   }
129 ] }
```

An abridged version of the tokenizer's initial, phase 0, output file.

In the previous listing an abridged version of the tokenizer's output is shown. This is the baseline JSON data structure we work with. The source attributes shown, although abridged, contain the source `ArtifactLine` and `Artifact` objects.

*Listing 19-17. AssemblerThumb.java Demonstration 5*

```
//output_tokened_phase1_valid_lines.json
001 {
002 "obj_name": "TokenLine",
003 "byteLength": 2,
004 "addressInt": 0,
005 "lineNumActive": 0,
006 "lineNumAbs": 5,
007 "payloadLen": 6,
008 "payloadLenArg": 0,
009 "isLineEmpty": false,
010 "isLineOpCode": false,
011 "isLineDirective": false,
012 "isLineLabelDef": false,
013 "source": { ... },
014 "validLineEntry": { ... },
015 "payload": [
016     {
017         "obj_name": "Token",
018         "type_name": "OpCode",
019         "source": "LDR",
020         "lineNumAbs": 5,
021         "index": 0,
022         "payloadLen": 0,
023         "payloadArgLen": 0,
024         "isOpCodeArg": false,
025         "isDirectiveArg": false,
026         "isOpCode": true,
027         "isDirective": false,
028         "isComment": false,
029         "isLabel": false,
030         "isLabelRef": false,
```

```
031     "artifact": { ... },
032     "type": { ... },
033     "payload": []
034   },
035   {
036     "obj_name": "Token",
037     "type_name": "RegisterLow",
038     "source": "R0",
039     "lineNumAbs": 5,
040     "index": 1,
041     "payloadLen": 0,
042     "payloadArgLen": 0,
043     "isOpCodeArg": false,
044     "isDirectiveArg": false,
045     "isOpCode": false,
046     "isDirective": false,
047     "isComment": false,
048     "isLabel": false,
049     "isLabelRef": false,
050     "artifact": { ... },
051     "type": { ... },
052     "payload": []
053   },
054   {
055     "obj_name": "Token",
056     "type_name": "GroupStart",
057     "source": "[",
058     "lineNumAbs": 5,
059     "index": 2,
060     "payloadLen": 0,
061     "payloadArgLen": 0,
062     "isOpCodeArg": false,
063     "isDirectiveArg": false,
064     "isOpCode": false,
065     "isDirective": false,
066     "isComment": false,
067     "isLabel": false,
068     "isLabelRef": false,
069     "artifact": { ... },
070     "type": { ... },
071     "payload": []
072   },
073   {
```

```
074     "obj_name": "Token",
075     "type_name": "RegisterPc",
076     "source": "PC",
077     "lineNumAbs": 5,
078     "index": 3,
079     "payloadLen": 0,
080     "payloadArgLen": 0,
081     "isOpCodeArg": false,
082     "isDirectiveArg": false,
083     "isOpCode": false,
084     "isDirective": false,
085     "isComment": false,
086     "isLabel": false,
087     "isLabelRef": false,
088     "artifact": { ... },
089     "type": { ... },
090     "payload": []
091   },
092   {
093     "obj_name": "Token",
094     "type_name": "Number",
095     "source": "#0",
096     "lineNumAbs": 5,
097     "index": 4,
098     "payloadLen": 0,
099     "payloadArgLen": 0,
100     "isOpCodeArg": false,
101     "isDirectiveArg": false,
102     "isOpCode": false,
103     "isDirective": false,
104     "isComment": false,
105     "isLabel": false,
106     "isLabelRef": false,
107     "artifact": { ... },
108     "type": { ... },
109     "payload": []
110   },
111   {
112     "obj_name": "Token",
113     "type_name": "GroupStop",
114     "source": "]",
115     "lineNumAbs": 5,
116     "index": 5,
```

```

117     "payloadLen": 0,
118     "payloadArgLen": 0,
119     "isOpCodeArg": false,
120     "isDirectiveArg": false,
121     "isOpCode": false,
122     "isDirective": false,
123     "isComment": false,
124     "isLabel": false,
125     "isLabelRef": false,
126     "artifact": { ... },
127     "type": { ... },
128     "payload": []
129   }
130 ]

```

The contents of the phase 1 output file showing addition of valid line data. This listing has been abridged for readability.

In the second output file we have now validated the line data associated with the token line against the instructions set's list of valid lines.

*Listing 19-18. AssemblerThumb.java Demonstration 6*

```

//output_tokened_phase2_refactored.json
001 {
002   "obj_name": "TokenLine",
003   "payloadOpCode": "LDR",
004   "byteLength": 2,
005   "addressHex": "0x00",
006   "addressBin": "0000000000000000",
007   "addressInt": 0,
008   "lineNumActive": 0,
009   "lineNumAbs": 5,
010   "payloadLen": 3,
011   "payloadLenArg": 3,
012   "isLineEmpty": false,
013   "isLineOpCode": true,
014   "isLineDirective": false,
015   "isLineLabelDef": false,
016   "source": { ... },

```

```
017 "validLineEntry": {
018     "obj_name": "is_valid_line",
019     "index": 5,
020     "empty_line": false,
021     "is_valid_line": [ ... ],
022     "name": ... ,
023     "fileName": ... ,
024     "loader": ...
025 },
026 "matchesOpCode": [
027     {
028         "obj_name": "is_op_code",
029         "op_code_name": "LDR",
030         "index": 38,
031         "arg_separator": ",",
032         "arg_len": 3,
033         "is_write_op_code": false,
034         "bit_rep": { ... },
035         "bit_series": { ... },
036         "args": [ ... ],
037         "name": ... ,
038         "fileName": ... ,
039         "loader": ...
040     },
041     {
042         "obj_name": "is_op_code",
043         "op_code_name": "LDR",
044         "index": 41,
045         "arg_separator": ",",
046         "arg_len": 3,
047         "is_write_op_code": false,
048         "bit_rep": { ... },
049         "bit_series": { ... },
050         "args": [ ... ],
051         "name": ... ,
052         "fileName": ... ,
053         "loader": ...
054     },
055     {
056         "obj_name": "is_op_code",
057         "op_code_name": "LDR",
058         "index": 48,
059         "arg_separator": ",",
```

```
060     "arg_len": 3,
061     "is_write_op_code": false,
062     "bit_rep": { ... },
063     "bit_series": { ... },
064     "args": [ ... ],
065     "name": ... ,
066     "fileName": ... ,
067     "loader": ...
068   },
069   {
070     "obj_name": "is_op_code",
071     "op_code_name": "LDR",
072     "index": 54,
073     "arg_separator": ",",
074     "arg_len": 3,
075     "is_write_op_code": false,
076     "bit_rep": { ... },
077     "bit_series": { ... },
078     "args": [ ... ],
079     "name": ... ,
080     "fileName": ... ,
081     "loader": ...
082   }
083 ],
084 "payload": [
085   {
086     "obj_name": "Token",
087     "type_name": "OpCode",
088     "source": "LDR",
089     "lineNumAbs": 5,
090     "index": 0,
091     "payloadLen": 0,
092     "payloadArgLen": 0,
093     "isOpCodeArg": false,
094     "isDirectiveArg": false,
095     "isOpCode": true,
096     "isDirective": false,
097     "isComment": false,
098     "isLabel": false,
099     "isLabelRef": false,
100     "artifact": { ... },
101     "type": { ... },
102     "payload": []
```

```
103     },
104     {
105         "obj_name": "Token",
106         "type_name": "RegisterLow",
107         "source": "R0",
108         "lineNumAbs": 5,
109         "index": 1,
110         "payloadLen": 0,
111         "payloadArgLen": 0,
112         "isOpCodeArg": true,
113         "isDirectiveArg": false,
114         "isOpCode": false,
115         "isDirective": false,
116         "isComment": false,
117         "isLabel": false,
118         "isLabelRef": false,
119         "artifact": { ... },
120         "type": { ... },
121         "register": { ... },
122         "payload": []
123     },
124     {
125         "obj_name": "Token",
126         "type_name": "GroupStart",
127         "source": "[",
128         "lineNumAbs": 5,
129         "index": 2,
130         "payloadLen": 3,
131         "payloadArgLen": 0,
132         "isOpCodeArg": true,
133         "isDirectiveArg": false,
134         "isOpCode": false,
135         "isDirective": false,
136         "isComment": false,
137         "isLabel": false,
138         "isLabelRef": false,
139         "artifact": { ... },
140         "type": { ... },
141         "payload": [
142             {
143                 "obj_name": "Token",
144                 "type_name": "RegisterPc",
145                 "source": "PC",
```

```
146         "lineNumAbs": 5,
147         "index": 0,
148         "payloadLen": 0,
149         "payloadArgLen": 0,
150         "isOpCodeArg": true,
151         "isDirectiveArg": false,
152         "isOpCode": false,
153         "isDirective": false,
154         "isComment": false,
155         "isLabel": false,
156         "isLabelRef": false,
157         "artifact": { ... },
158         "type": { ... },
159         "register": { .. },
160         "payload": []
161     },
162     {
163         "obj_name": "Token",
164         "type_name": "Number",
165         "source": "#0",
166         "lineNumAbs": 5,
167         "index": 1,
168         "payloadLen": 0,
169         "payloadArgLen": 0,
170         "isOpCodeArg": true,
171         "isDirectiveArg": false,
172         "isOpCode": false,
173         "isDirective": false,
174         "isComment": false,
175         "isLabel": false,
176         "isLabelRef": false,
177         "artifact": { ... },
178         "type": { ... },
179         "payload": []
180     },
181     {
182         "obj_name": "Token",
183         "type_name": "GroupStop",
184         "source": "]",
185         "lineNumAbs": 5,
186         "index": 2,
187         "payloadLen": 0,
188         "payloadArgLen": 0,
```

```

189         "isOpCodeArg": true,
190         "isDirectiveArg": false,
191         "isOpCode": false,
192         "isDirective": false,
193         "isComment": false,
194         "isLabel": false,
195         "isLabelRef": false,
196         "artifact": { ... },
197         "type": { ... },
198         "payload": []
199     }
200 ]
201 }
202 ]

```

The contents of the phase 2 output file showing addition of opcode matches and the refactoring of the opcode group's argument tokens. This listing has been abridged for readability.

A few major changes occur in this version of the JSON data structure. The token line now has an opcode definition associated with it, including all opcode arguments, and the original token structure has been adjusted to use sub-tokens for the group opcode argument.

*Listing 19-19. AssemblerThumb.java Demonstration 7*

```
//output_tokened_phase3_valid_lines.json
001 {
002 "obj_name": "TokenLine",
003 "payloadOpCode": "LDR",
004 "byteLength": 2,
005 "addressHex": "0x00",
006 "addressBin": "0000000000000000",
007 "addressInt": 0,
008 "lineNumActive": 0,
009 "lineNumAbs": 5,
010 "payloadLen": 3,
011 "payloadLenArg": 3,
012 "isLineEmpty": false,
013 "isLineOpCode": true,
```

```
014 "isLineDirective": false,
015 "isLineLabelDef": false,
016 "source": { ... },
017 "validLineEntry": {
018     "obj_name": "is_valid_line",
019     "index": 5,
020     "empty_line": false,
021     "is_valid_line": [ ... ],
022     "name": ... ,
023     "fileName": ... ,
024     "loader": ...
025 },
026 "matchesOpCode": [
027     {
028         "obj_name": "is_op_code",
029         "op_code_name": "LDR",
030         "index": 38,
031         "arg_separator": ",",
032         "arg_len": 3,
033         "is_write_op_code": false,
034         "bit_rep": { ... },
035         "bit_series": { ... },
036         "args": [ ... ],
037         "name": ... ,
038         "fileName": ... ,
039         "loader": ...
040     }
041 ],
042 "payload": [
043     {
044         "obj_name": "Token",
045         "type_name": "OpCode",
046         "source": "LDR",
047         "lineNumAbs": 5,
048         "index": 0,
049         "payloadLen": 0,
050         "payloadArgLen": 0,
051         "isOpCodeArg": false,
052         "isDirectiveArg": false,
053         "isOpCode": true,
054         "isDirective": false,
055         "isComment": false,
056         "isLabel": false,
```

```
057    "isLabelRef": false,
058    "artifact": { ... },
059    "type": { ... },
060    "payload": []
061  },
062  {
063    "obj_name": "Token",
064    "type_name": "RegisterLow",
065    "source": "R0",
066    "lineNumAbs": 5,
067    "index": 1,
068    "payloadLen": 0,
069    "payloadArgLen": 0,
070    "isOpCodeArg": true,
071    "isDirectiveArg": false,
072    "isOpCode": false,
073    "isDirective": false,
074    "isComment": false,
075    "isLabel": false,
076    "isLabelRef": false,
077    "artifact": { ... },
078    "type": { ... },
079    "register": { ... },
080    "payload": []
081  },
082  {
083    "obj_name": "Token",
084    "type_name": "GroupStart",
085    "source": "[",
086    "lineNumAbs": 5,
087    "index": 2,
088    "payloadLen": 3,
089    "payloadArgLen": 0,
090    "isOpCodeArg": true,
091    "isDirectiveArg": false,
092    "isOpCode": false,
093    "isDirective": false,
094    "isComment": false,
095    "isLabel": false,
096    "isLabelRef": false,
097    "artifact": { ... },
098    "type": { ... },
099    "payload": [
```

```

100      {
101          "obj_name": "Token",
102          "type_name": "RegisterPc",
103          "source": "PC",
104          "lineNumAbs": 5,
105          "index": 0,
106          "payloadLen": 0,
107          "payloadArgLen": 0,
108          "isOpCodeArg": true,
109          "isDirectiveArg": false,
110          "isOpCode": false,
111          "isDirective": false,
112          "isComment": false,
113          "isLabel": false,
114          "isLabelRef": false,
115          "artifact": { ... },
116          "type": { ... },
117          "register": { ... },
118          "payload": []
119      },
120      {
121          "obj_name": "Token",
122          "type_name": "Number",
123          "source": "#0",
124          "lineNumAbs": 5,
125          "index": 1,
126          "payloadLen": 0,
127          "payloadArgLen": 0,
128          "isOpCodeArg": true,
129          "isDirectiveArg": false,
130          "isOpCode": false,
131          "isDirective": false,
132          "isComment": false,
133          "isLabel": false,
134          "isLabelRef": false,
135          "artifact": { ... },
136          "type": { ... },
137          "payload": []
138      },
139      {
140          "obj_name": "Token",
141          "type_name": "GroupStop",
142          "source": "]",

```

```

143         "lineNumAbs": 5,
144         "index": 2,
145         "payloadLen": 0,
146         "payloadArgLen": 0,
147         "isOpCodeArg": true,
148         "isDirectiveArg": false,
149         "isOpCode": false,
150         "isDirective": false,
151         "isComment": false,
152         "isLabel": false,
153         "isLabelRef": false,
154         "artifact": { ... },
155         "type": { ... },
156         "payload": []
157     }
158 ]
159 }
160 ] }

```

A section of output from the `output_tokened_phase3_valid_lines.json` file.

In the previous listing, at this point in the permutation of the JSON data structure, we've determined an exact valid line match and completed all data structure permutations like collapsing comments, lists, and groups while expanding register ranges.

*Listing 19-20. AssemblerThumb.java Demonstration 8*

```

//output_tokened_phase4_bin_output.json
001 {
002 "obj_name": "TokenLine",
003 "payloadOpCode": "LDR",
004 "payloadBinRepStrEndianBig1": "0100100000000000",
005 "payloadBinRepStrEndianLill1": "000000001001000",
006 "byteLength": 2,
007 "addressHex": "0x00",
008 "addressBin": "0000000000000000",
009 "addressInt": 0,
010 "lineNumActive": 0,
011 "lineNumAbs": 5,

```

```
012 "payloadLen": 3,
013 "payloadLenArg": 3,
014 "isLineEmpty": false,
015 "isLineOpCode": true,
016 "isLineDirective": false,
017 "isLineLabelDef": false,
018 "source": { ... },
019 "validLineEntry": { ... },
020 "matchesOpCode": [
021   {
022     "obj_name": "is_op_code",
023     "op_code_name": "LDR",
024     "index": 38,
025     "arg_separator": ",",
026     "arg_len": 3,
027     "is_write_op_code": false,
028     "bit_rep": { ... },
029     "bit_series": { ... },
030     "args": [ ... ],
031     "name": ... ,
032     "fileName": ... ,
033     "loader": ...
034   }
035 ],
036 "buildEntries": [
037   {
038     "obj_name": "BuildOpCodeThumb",
039     "bitSeries": { ... },
040     "opCode": { ... },
041     "tokenOpCode": { ... },
042     "isOpCode": true,
043     "isOpCodeArg": false,
044     "isOpCodeArgGroup": false,
045     "isOpCodeArgList": false,
046     "binRepStr1": "01001"
047   },
048   {
049     "obj_name": "BuildOpCodeThumb",
050     "bitSeries": { ... },
051     "opCodeArg": { ... },
052     "tokenOpCodeArg": { ... },
053     "isOpCode": false,
054     "isOpCodeArg": true,
```

```

055     "isOpCodeArgGroup": false,
056     "isOpCodeArgList": false,
057     "binRepStr1": "000"
058   },
059   {
060     "obj_name": "BuildOpCodeThumb",
061     "bitSeries": { ... },
062     "opCodeArg": { ... },
063     "tokenOpCodeArg": { ... },
064     "isOpCode": false,
065     "isOpCodeArg": true,
066     "isOpCodeArgGroup": false,
067     "isOpCodeArgList": false,
068     "binRepStr1": ""
069   },
070   {
071     "obj_name": "BuildOpCodeThumb",
072     "bitSeries": { ... },
073     "opCodeArgGroup": { ... },
074     "tokenOpCodeArgGroup": { ... },
075     "isOpCode": false,
076     "isOpCodeArg": false,
077     "isOpCodeArgGroup": true,
078     "isOpCodeArgList": false,
079     "binRepStr1": "00000000"
080   },
081   {
082     "obj_name": "BuildOpCodeThumb",
083     "bitSeries": { ... },
084     "opCodeArgGroup": { ... },
085     "tokenOpCodeArgGroup": { ... },
086     "isOpCode": false,
087     "isOpCodeArg": false,
088     "isOpCodeArgGroup": true,
089     "isOpCodeArgList": false,
090     "binRepStr1": ""
091   }
092 ],
093 "payload": [
094   {
095     "obj_name": "Token",
096     "type_name": "OpCode",
097     "source": "LDR",

```

```
098     "lineNumAbs": 5,
099     "index": 0,
100     "payloadLen": 0,
101     "payloadArgLen": 0,
102     "isOpCodeArg": false,
103     "isDirectiveArg": false,
104     "isOpCode": true,
105     "isDirective": false,
106     "isComment": false,
107     "isLabel": false,
108     "isLabelRef": false,
109     "artifact": { ... },
110     "type": { ... },
111     "payload": []
112   },
113   {
114     "obj_name": "Token",
115     "type_name": "RegisterLow",
116     "source": "R0",
117     "lineNumAbs": 5,
118     "index": 1,
119     "payloadLen": 0,
120     "payloadArgLen": 0,
121     "isOpCodeArg": true,
122     "isDirectiveArg": false,
123     "isOpCode": false,
124     "isDirective": false,
125     "isComment": false,
126     "isLabel": false,
127     "isLabelRef": false,
128     "artifact": { ... },
129     "type": { ... },
130     "register": { ... },
131     "payload": []
132   },
133   {
134     "obj_name": "Token",
135     "type_name": "GroupStart",
136     "source": "[",
137     "lineNumAbs": 5,
138     "index": 2,
139     "payloadLen": 3,
140     "payloadArgLen": 0,
```

```
141     "isOpCodeArg": true,
142     "isDirectiveArg": false,
143     "isOpCode": false,
144     "isDirective": false,
145     "isComment": false,
146     "isLabel": false,
147     "isLabelRef": false,
148     "artifact": { ... },
149     "type": { ... },
150     "payload": [
151       {
152         "obj_name": "Token",
153         "type_name": "RegisterPc",
154         "source": "PC",
155         "lineNumAbs": 5,
156         "index": 0,
157         "payloadLen": 0,
158         "payloadArgLen": 0,
159         "isOpCodeArg": true,
160         "isDirectiveArg": false,
161         "isOpCode": false,
162         "isDirective": false,
163         "isComment": false,
164         "isLabel": false,
165         "isLabelRef": false,
166         "artifact": { ... },
167         "type": { ... },
168         "register": { ... },
169         "payload": []
170       },
171       {
172         "obj_name": "Token",
173         "type_name": "Number",
174         "source": "#0",
175         "value": 0,
176         "lineNumAbs": 5,
177         "index": 1,
178         "payloadLen": 0,
179         "payloadArgLen": 0,
180         "isOpCodeArg": true,
181         "isDirectiveArg": false,
182         "isOpCode": false,
183         "isDirective": false,
```

```

184         "isComment": false,
185         "isLabel": false,
186         "isLabelRef": false,
187         "artifact": { ... },
188         "type": { ... },
189         "payload": []
190     },
191     {
192         "obj_name": "Token",
193         "type_name": "GroupStop",
194         "source": "]",
195         "lineNumAbs": 5,
196         "index": 2,
197         "payloadLen": 0,
198         "payloadArgLen": 0,
199         "isOpCodeArg": true,
200         "isDirectiveArg": false,
201         "isOpCode": false,
202         "isDirective": false,
203         "isComment": false,
204         "isLabel": false,
205         "isLabelRef": false,
206         "artifact": { ... },
207         "type": { ... },
208         "payload": []
209     }
210   ]
211 }
212 ] }

```

A section of output from the `output_tokened_phase4_valid_lines.json` file.

In the last version of the JSON data structure, we have a list of build entries, and we have binary encoding data associated with the token line. Notice that we're basically looking at the structure as it exists in the assembler's class fields and variables. This is the benefit of working with JSON data, it's both a file and an object that can express structure. That brings us to the conclusion of the `AssemblerThumb` class' review!

# Chapter Conclusion

We covered a lot of material in this chapter. Also, we got a very detailed look at what the assembler is doing underneath the hood. Let's take a look at the topics we covered in this chapter.

- **Reviewed Step 11:** In Step 11 we perform the final validation of the opcode lines of the assembly program. To find an exact match we compare the loaded description of the opcode and its arguments, loaded from the instruction set data files, with the token line and token data loaded from the assembly source code.
- **Reviewed Step 12:** In Step 12 we perform the final validation of the directive lines of the assembly program. We perform a similar argument comparison to find the correct directive match.
- **Reviewed Step 13:** In Step 13 the assembler builds the binary representation of the directive and opcode lines. We saw how the assembler creates binary representations of each token and aggregates them to create a representation for the token line.
- **Demonstrated Permutation of the Data Structure:** We took a deep dive into the JSON data structure that represents the state of the assembler as it processes the assembly source code.

At this point in the text, you have looked at just about every line of code in the GenAsm project and should have a solid understanding of how the assembler works. In the next chapter we'll take a look at the last class in the book that we'll have to review, the linker.

# **Chapter 20: The Linker**

Welcome to Chapter 20, the last class review chapter in the text. In this chapter we'll take a look at the simple linker that's included with the GenAsm project. This is by no means a sophisticated linker. In fact, it's simply designed to write the binary representation of all the token lines associated with a code area, a data area, or both.

The linker takes care of a few things. The main responsibility is linking the two areas together in one linear block of assembly source code. It's also responsible for writing the listing and binary output files for both big- and littler-endian byte encodings. In this chapter we'll take a look at the Thumb-1 implementation of the `Linker` interface. We'll employ the review sections listed here.

- Class Fields
- Pertinent Method Outline/Class Headers
- Support Method Details
- Main Method Details
- Demonstration

There are no static class fields, or pertinent enumerations to speak of, so we'll omit those sections. This is the last class we have to review, let's get to it!

## Class Review: LinkerThumb.java

The `LinkerThumb` class is an important step in the assembly process. Its job is to connect the different pieces of our assembly program into one coherent binary output file. Due to the simplicity of our assembler we only have to worry about managing up to two areas, one code, and one data. We'll start off the review in the next section with a listing of the class' fields.

## Class Fields: LinkerThumb.java

The `LinkerThumb` class has a few fields for us to review, listed as follows:

*Listing 20-1. LinkerThumb.java Class Fields*

```
public boolean verboseLogs = false;
public boolean quellFileOutput = false;
public ArrayList<Byte> binBe = null;
public ArrayList<Byte> binLe = null;
public Hashtable<String, String> hexMapBe = null;
public Hashtable<String, String> hexMapLe = null;
```

A list of the `LinkerThumb` class fields.

The first two fields listed are Boolean flags that toggle verbose logging and file output. The next two entries, `binBe` and `binLe`, are `ArrayList`s that are meant to hold the byte values that will be written to the output files. Lastly, there are two fields, `hexMapBe` and `hexMapLe`, that are responsible for mapping an active line number to the hexadecimal representation for that line. In the next section we'll take a look at the class' declaration and pertinent methods.

# Pertinent Method Outline/Class Headers: LinkerThumb.java

The LinkerThumb class has the following pertinent methods up for review.

## *Listing 20-2. LexerThumb.java Pertinent Method Outline/Class Headers 1*

```
//Support Methods
public int AddBytes(String binStr, ArrayList<Byte> data);

//Main Methods
public void RunLinker(Assembler assembler, String assemblySourceFile,
String outputDir, Object otherObj, boolean verbose, boolean quellOutput);
```

The pertinent support and main methods of the LinkerThumb class.

Not too bad, only one in each category. In the next listing the class declaration, including import statements and interfaces, is shown.

## *Listing 20-3. LexerThumb.java Pertinent Method Outline/Class Headers 2*

```
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.Hashtable;
import java.util.List;
import java.util.Map;
import net.middlemind.GenAsm.AssemblersAssembler;
import net.middlemind.GenAsm.AssemblersSymbol;
import net.middlemind.GenAsm.AssemblersThumbAssemblerThumb;
import net.middlemind.GenAsm.FileIO.FileUnloader;
import net.middlemind.GenAsm.LinkersLinker;
import net.middlemind.GenAsm.Logger;
import net.middlemind.GenAsm.TokenersTokenLine;
import net.middlemind.GenAsm.Utils;

public class LinkerThumb implements Linker { }
```

A listing of the LinkerThumb class' declaration.

Note that the `LinkerThumb` class implements the `Linker` interface and as such must define the `RunLinker` method. Let's take a look at the class' support methods next.

## Support Method Details: LinkerThumb.java

The `LinkerThumb` class has only one support method to speak of, listed here.

*Listing 20-4. LinkerThumb.java Support Method Details 1*

```
01 public int AddBytes(String binStr, ArrayList<Byte> data) {  
02     int len = binStr.length() / 8;  
03     String s = null;  
04     Byte b = null;  
05     int ret = 0;  
06     for (int i = 0; i < len; i++) {  
07         s = binStr.substring((i * 8), (i * 8) + 8);  
08         b = (byte) Integer.parseInt(s, 2);  
09         data.add(b);  
10         ret++;  
11     }  
12     return ret;  
13 }
```

The `AddBytes` support method of the `LinkerThumb` class.

The `AddBytes` method is used to append byte data from the `binStr` argument to the `ArrayList`, `data` argument. To do so the binary string is split into 8-character sub-strings using the length determined on line 2. We iterate over the `binStr` argument pulling out a byte of characters, line 7. This value is converted to a byte and stored in the `data` list on line 9. The method returns the number of bytes added in this way.

## Main Method Details: LinkerThumb.java

The RunLinker method is fairly long, but we have the benefit of it being inherently redundant. The method is designed to process both big- and little-endian versions of the assembly program. This means it has to create output files for both byte level encodings. The benefit here is that the first half of the method is pretty much repeated for the second encoding. As such, we only have to review the first half of the method. I've broken the review into four parts, listed subsequently.

*Listing 20-5. LinkerThumb.java Main Method Details 1*

```
001 public void RunLinker(Assembler assembler, String assemblySourceFile,
String outputDir, Object otherObj, boolean verbose, boolean quellOutput)
throws Exception {
002     Logger.wrl("LinkerThumb: RunLinker");
003     verboseLogs = verbose;
004     quellFileOutput = quellOutput;
005     AssemblerThumb asm = (AssemblerThumb) assembler;
006     Map<Integer, TokenLine> fin = new Hashtable<>();
007     List<String> lstFile = new ArrayList<>();
008     TokenLine line = null;
009
010     if (asm.areaThumbCode != null && asm.areaThumbData != null) {
011         if (asm.areaThumbCode.lineNumEntry <
asm.areaThumbData.lineNumEntry) {
012             //code first
013             for (int i = asm.areaThumbCode.lineNumEntry + 1; i <
asm.areaThumbCode.lineNumEnd; i++) {
014                 line = asm.asmDataTokened.get(i);
015                 if (!line.isEmpty && !line.isLabelDef) {
016                     fin.put(line.lineNumAbs, line);
017                 }
018             }
019
020             for (int i = asm.areaThumbData.lineNumEntry + 1; i <
asm.areaThumbData.lineNumEnd; i++) {
021                 line = asm.asmDataTokened.get(i);
022                 if (!line.isEmpty && line.isDirective && !
line.isLabelDef) {
023                     fin.put(line.lineNumAbs, line);
```

```
024         }
025     }
026 } else {
027     //data first
028     for (int i = asm.areaThumbData.lineNumEntry + 1; i <
asm.areaThumbData.lineNumEnd; i++) {
029         line = asm.asmDataTokened.get(i);
030         if (!line.isEmpty && line.isLineDirective && !
line.isLineLabelDef) {
031             fin.put(line.lineNumAbs, line);
032         }
033     }
034
035     for (int i = asm.areaThumbCode.lineNumEntry + 1; i <
asm.areaThumbCode.lineNumEnd; i++) {
036         line = asm.asmDataTokened.get(i);
037         if (!line.isEmpty && !line.isLineLabelDef) {
038             fin.put(line.lineNumAbs, line);
039         }
040     }
041 }
042
043 } else if (asm.areaThumbCode != null) {
044     for (int i = asm.areaThumbCode.lineNumEntry + 1; i <
asm.areaThumbCode.lineNumEnd; i++) {
045         line = asm.asmDataTokened.get(i);
046         if (!line.isEmpty && !line.isLineLabelDef) {
047             fin.put(line.lineNumAbs, line);
048         }
049     }
050
051 } else if (asm.areaThumbData != null) {
052     for (int i = asm.areaThumbData.lineNumEntry + 1; i <
asm.areaThumbData.lineNumEnd; i++) {
053         line = asm.asmDataTokened.get(i);
054         if (!line.isEmpty && line.isLineDirective && !
line.isLineLabelDef) {
055             fin.put(line.lineNumAbs, line);
056         }
057     }
058 }
059 }
```

```

060     Logger.wrl("LinkerThumb: RunLinker: Found " + fin.size() + " lines
of linked assembly json objects");
061     Utils.WriteObject(fin, "Linked Assembly Source Code Lines",
"output_linked_area_lines_code.json", outputDir);
062 ...

```

Part 1 of the LinkerThumb class' RunLinker method showing preparation of active lines.

The first two entries in the method, lines 3 – 4, are used to set Boolean flags in the class fields as shown. A reference to the assembler is stored locally in the `asm` variables, line 5. The `fin` variable is a Map from an Integer to a TokenLine and it's used to hold the final list of token lines to process, active token lines. Following this entry is the `lstFile` which is a List designed to hold the contents of the output files. Lastly, the temporary variable, `line`, is used to hold a reference to the current line we're working on.

The code on lines 10 – 43 is used to populate the `fin` variable with all the token lines that will be present in the final binary representation of the program. The linker supports the case where the code area comes first, lines 12 – 25, and the case where it comes second, lines 27 – 40. The remaining cases where only the code area is defined, line 43 – 49, and when only the data area is defined, lines 51 – 58. An output file showing which lines have been included is written on line 61. Let's take a look at the second listing in this method review.

#### ***Listing 20-6. LinkerThumb.java Main Method Details 2***

```

063     int lineNumberOff = 0;
064     int addressIntOff = 0;
065     String cleanHexAddr = null;
066     String tmp1 = null;
067     String tmp2 = null;
068     TokenLine tmpLine = null;
069     int count = 0;
070     int prevLineNumAbs = -1;
071     binBe = new ArrayList<>();
072     binLe = new ArrayList<>();

```

```

073     hexMapBe = new Hashtable<>();
074     hexMapLe = new Hashtable<>();
075     int totalBytes = 0;
076     boolean addToHexMapOn = false;
077
078     for (int i = 0; i < asm.asmDataTokened.size(); i++) {
079         tmpLine = asm.asmDataTokened.get(i);
080         tmp1 = tmpLine.lineNumAbs + lineNumberOff + "";
081         tmp1 = Utils.FormatBinString(tmp1, 10, true);
082         tmp1 += "\t";
083         tmp1 += Utils.FormatBinString(tmpLine.addressInt + addressIntOff
+ "", 10, true);
084         tmp1 += "\t";
085         tmp2 = "";
086         if (fin.containsKey(tmpLine.lineNumAbs) == true &&
tmpLine.isEmpty == false) {
087             if (tmpLine.payloadBinRepStrEndianBig1 != null) {
088                 totalBytes += AddBytes(tmpLine.payloadBinRepStrEndianBig1,
binBe);
089                 cleanHexAddr =
Utils.CleanHexPrefix(Utils.FormatHexString(tmpLine.addressHex,
asm.lineLenBytes * 4, true));
090                 if (cleanHexAddr.equals("000000F4") == true ||
cleanHexAddr.equals("080000F4") == true) {
091                     addToHexMapOn = true;
092                 }
093
094                 if (addToHexMapOn == true) {
095                     hexMapBe.put(cleanHexAddr,
Utils.CleanHexPrefix(Utils.FormatHexString(Utils.Bin2Hex(tmpLine.payloadBi
nRepStrEndianBig1), asm.lineLenBytes * 2, true)));
096                 }
097                 tmp1 += Utils.FormatBinString(tmpLine.lineNumActive + "",
10, true) + "\t" + Utils.FormatHexString(tmpLine.addressHex,
asm.lineLenBytes * 4, true) + "\t" +
Utils.PrettyBin(tmpLine.payloadBinRepStrEndianBig1,
asm.jsonObjIsOpCodes.bit_series.bit_len, true) + "\t" +
Utils.PrettyHex(Utils.Bin2Hex(tmpLine.payloadBinRepStrEndianBig1),
asm.lineLenBytes * 2, true) + "\t" + tmpLine.source.source;
098             } else {
099                 if (!Utils.IsEmpty(tmpLine.addressHex)) {
100                     tmp1 += Utils.FormatBinString(tmpLine.lineNumActive +
"", 10, true) + "\t" + Utils.FormatHexString(tmpLine.addressHex,

```

```

asm.lineLenBytes * 4, true) + "\t" + "          " + "\t\t\t\t" +
tmpLine.source.source;
101           } else {
102               tmp1 += Utils.FormatBinString(tmpLine.lineNumActive +
103 "", 10, true) + "\t          \t" + "          " + "\t\t\t\t" +
tmpLine.source.source;
104           }
105
106           if (Utils.IsEmpty(tmpLine.payloadBinRepStrEndianBig2)
107 == false) {
108               //handle second line
109               ...
110           } else {
111               tmp1 += "    \t    \t          \t" +
tmpLine.source.source;
112           }
113
114           if (prevLineNumAbs != -1 && tmpLine.lineNumAbs !=
115 (prevLineNumAbs + 1)) {
116               tmp1 += " #####ERROR";
117           }
118
119           prevLineNumAbs = tmpLine.lineNumAbs;
120           lstFile.add(tmp1);
121           count++;
122           if (Utils.IsEmpty(tmp2) == false) {
123               lstFile.add(tmp2);
124               count++;
125           }
126       }
127   }
128
129   ...

```

Part 2 of the LinkerThumb class' RunLinker method showing an abridged version of the creation of a line of the listing file.

Part 2 of the RunLinker method looks complicated but it's actually straightforward. The code shown in the previous listing creates different lines of the listing file, indicating if a line of assembly source code is an active line

or not. Take a look at the code and make sure you understand it before moving on to the next section.

*Listing 20-7. LinkerThumb.java Main Method Details 3*

```
145     lstFile.add("");
146     lstFile.add(";===== SYMBOL TABLE =====");
147     Symbol sym = null;
148     tmp1 = null;
149     for (String key : asm.symbols.symbols.keySet()) {
150         sym = asm.symbols.symbols.get(key);
151         if (sym.value != null) {
152             tmp1 = ";Name: " + sym.name + "\tLineNumAbs: " +
Utils.FormatBinString(sym.lineNumAbs + "", 10, true) + "\tLineNumActive: " +
Utils.FormatBinString(sym.lineNumActive + "", 10, true) + "\tAddressHex: " +
Utils.FormatHexString(sym.addressHex, asm.lineLenBytes * 4, true) + "\tValue: " +
sym.value.toString() + "\tEmptyLineLabel: " +
sym.isEmptyLineLabel + "\tIsLabel: " + sym.isLabel + "\tIsStaticValue: " +
sym.isStaticValue;
153         } else {
154             tmp1 = ";Name: " + sym.name + "\tLineNumAbs: " +
Utils.FormatBinString(sym.lineNumAbs + "", 10, true) + "\tLineNumActive: " +
Utils.FormatBinString(sym.lineNumActive + "", 10, true) + "\tAddressHex: " +
Utils.FormatHexString(sym.addressHex, asm.lineLenBytes * 4, true) + "\tValue: " +
"n/a" + "\tEmptyLineLabel: " + sym.isEmptyLineLabel + "\tIsLabel: " +
sym.isLabel + "\tIsStaticValue: " + sym.isStaticValue;
155         }
156         lstFile.add(tmp1);
157     }
158     ...
```

Part 3 of the LinkerThumb class' RunLinker method showing the listing file's symbol table for the current program.

The code in Part 3, shown previously, is responsible for adding a symbol table to the listing file generated by the linker. Take a moment to trace through the values printed for each symbol and make sure you understand what's going on here before moving on to the next section.

#### *Listing 20-8. LinkerThumb.java Main Method Details 4*

```
159     Logger.wrl("Found total bytes Big Endian: " + totalBytes);
160     int len = binBe.size();
161     byte[] data = new byte[len];
162     for (int i = 0; i < len; i++) {
163         data[i] = (byte) binBe.get(i);
164     }
165     lstFile.add("");
166     lstFile.add(";===== BINARY OUTPUT =====");
167     lstFile.add(";Expected file output size: " + data.length);
168
169     if (!quellFileOutput) {
170         FileUnloader.WriteList(Paths.get(outputDir,
171                                         "output_assembly_listing_endian_big.list").toString(), lstFile);
172         FileUnloader.WriteBuffer(Paths.get(outputDir,
173                                         "output_assembly_listing_endian_big.bin").toString(), data);
172     }
173     ...
174 }
```

Part 4 of the `LinkerThumb` class' `RunLinker` method showing the summary of the binary output and the writing of the listing and final program output files.

The last snippet of code from this method is responsible for appending a binary output summary to the listing file and writing out the final binary representation of the assembly program on lines 170 and 171. Again, this code is rather direct so we won't go into any detail reviewing it here. Please take a moment and read over it. Make sure you understand it before moving on. In the next section we'll take a look at the `LinkerThumb` class in action.

## Demonstration: `LinkerThumb.java`

To demonstrate the `LinkerThumb` class in action I wanted to show the contents of the list file generated as part of the assembly process.

#### *Listing 20-9. LinkerThumb.java Demonstration*

Program: TEST\_0\_AsmDemo

File: output\_assembly\_listing\_endian\_lil.list

```

01 0000000000    0000000000 @TTL |AssemblerDemo|
02 0000000001    0000000000 @SUBT |SubTitleTest|
03 0000000002    0000000000 @AREA |Program| @CODE, @READONLY
04 0000000003    0000000000 @ENTRY
05 0000000004    0000000000
06 0000000005    0000000000 0000000000 0x00000000 00000000 01001000
          0x00 48      LDR     R0, [PC, #0]
07 0000000006    0000000000
08 0000000007    0000000000 @END
09
10 ;===== SYMBOL TABLE =====
11
12 ;===== BINARY OUTPUT =====
13 ;Expected file output size: 2

```

The contents of the listing file for the TEST\_O\_AsmDemo program.

Take a close look at the linker class' listing output file. This output file is associated with running the TEST\_O\_AsmDemo program. I should mention that this program is not designed to run in the emulator. Its purpose is to demonstrate assembler internal functionality. There are a few programs like this included in the examples. We'll cover that in the next chapter when we review some of them. The difference between an active line of assembly source code, and one that is not, should be clear given the differences in their listing file entries. Take a moment to look at some of the listing file's output from other example programs. Be sure to look at files with big- and little-endian byte encodings.

## Chapter Conclusion

Congratulations! You've completed the entire code review of the GenAsm assembler project! In this chapter you got to take a look at a simple linker that plugs into the AssemblerThumb class and is responsible for generating a number of output files. Most notably are the listing files and final binary representations of the assembly program. In the next chapter we'll take a detailed look at some of the example programs included with the project.

# Chapter 21: The Example Programs

You made your way through a ton of documentation and code review material and your reward for such diligence is ... well more material. But!, this time the discussion will be light as we'll be talking about the project's example programs. We won't be doing a review of how to program in ARM Thumb-1 assembly, that is beyond the scope of this book. Let's start things off by taking a look at a list of the included example programs.

*Listing 21-1. List of Example Programs*

Program Name	Type	Executable
TEST_A_AllOpCodes	Assembler Validation	No
TEST_B_16BitXfer	Example	Yes
TEST_C_OnesComp	Example	Yes
TEST_D_16BitAdd	Example	Yes
TEST_E_SplitByte	Example	Yes
TEST_F_CompareNum	Example	Yes
TEST_G_64BitAdd	Example	Yes
TEST_H_ShiftLeft	Example	Yes
TEST_I_Factorial	Example	Yes
TEST_J_16BitSeriesAddOne	Example	Yes
TEST_K_16BitSeriesAddTwo	Example	Yes

TEST_L_HelloWorld	Example	Yes
TEST_M_YourName	Example	Yes
TEST_N_AsmChecks	Assembler Validation	No
TEST_O_AsmDemo	Assembler Validation	No

A descriptive listing of the included example and test assembly programs. Executable means the program is designed to run in a GameBoy Advance emulator.

There are quite a few of them. I should take a moment to make a special mention of Mr. Keith Akuma who was nice enough to provide me with a copy of his GameBoy Advance screen test program written in ARM Thumb-1 VASM assembly code. I was really stuck, not only working on an assembler, but trying to learn assembly itself. Faced with now having to master ARM 32-bit assembly opcodes and then port them back to Thumb-1 instructions, and convert that from VASM to my GenAsm syntax, needless to say was losing ground fast and facing difficult hurdles I wasn't sure I could overcome. I reached out to Mr. Keith Akuma and asked him for some help, and he was kind enough to help me, single-handedly saving this project from failure. Thank you again!

Keith Akuma

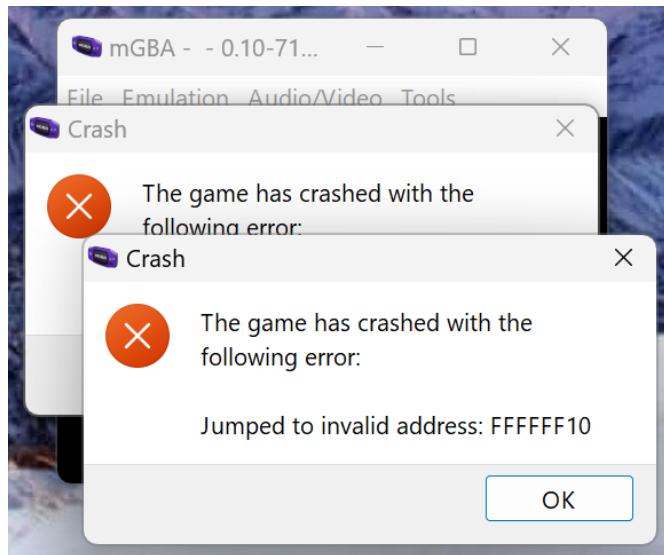
<https://www.chibiakumas.com/>

<https://www.assemblytutorial.com/>

I've provided some of Mr. Akuma's URLs here. Be sure to check them out, and don't forget to look for his books online. They are some of the best assembly programming books I've read. Now back to the list of example programs. The previous listing indicates if a given assembly program is meant to be executed with the GameBoy Advance emulator, noted by a value of yes in the "Executable" column. Some programs are meant only to be executed by the assembler to validate functionality and don't constitute a logical program.

In fact if you run one of the assembly programs that is not meant to be executed you may end up with some errors and an emulator that is going crazy.

*Image 21-1. Running a Bad Assembly Program in the Emulator*



An image showing the mGBA emulator crashing on an assembly program that is not designed to run in the GameBoy Advance emulator.

No worries. Just open up the task or process manager for the operating system you're using and kill the emulator process.

*Image 21-2. Stopping a Crashed Emulator in Windows*

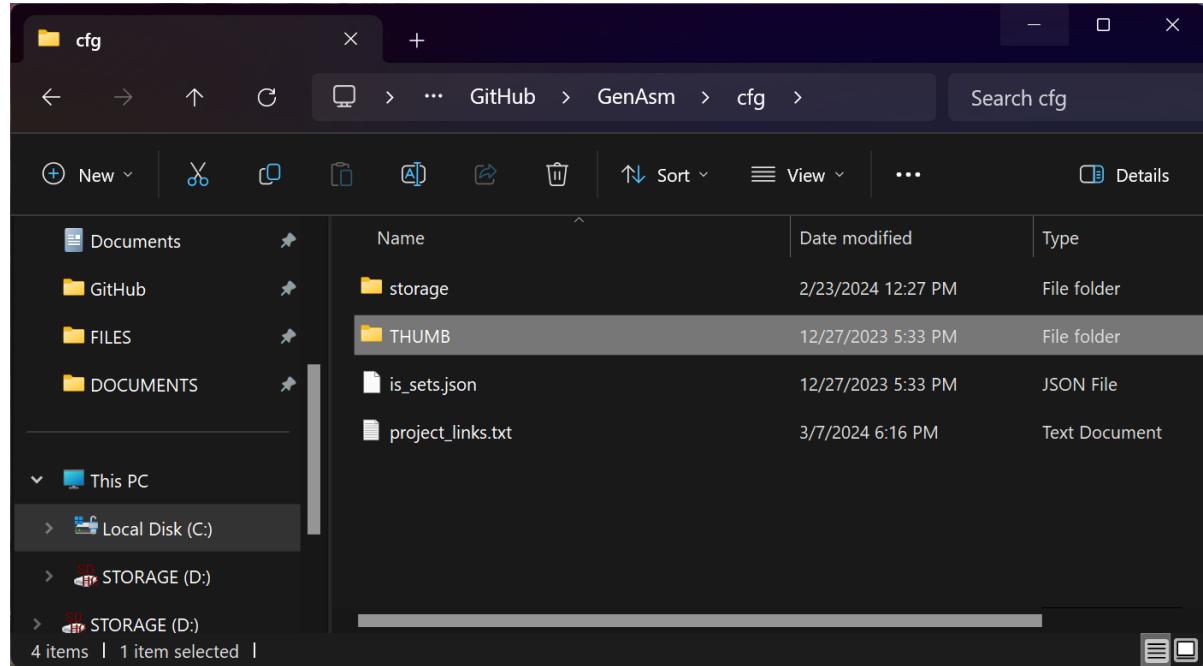
The screenshot shows the Windows Task Manager window titled "Task Manager". The main pane is labeled "Processes" and lists various running applications. The "mGBA Game Boy Advance" process is selected, showing its resource usage: 3.9% CPU, 371.5 MB Memory, 0 MB/s Disk, and 0 Mbps Network. A context menu is open over this process, with the "End task" option highlighted. Other options in the menu include "Expand", "Resource values", "Provide feedback", "Efficiency mode", "Debug", and "Create memory dump file". The Task Manager interface includes a search bar at the top and a toolbar with icons for running new tasks, ending tasks, and switching to efficiency mode.

Name	Status	CPU	Memory	Disk	Network
Google Chrome (31)		3.9%	371.5 MB	0 MB/s	0 Mbps
LibreOffice		0.1%	256.3 MB	0 MB/s	0 Mbps
mGBA Game Boy Advance		0.5%	35.9 MB	0 MB/s	0 Mbps
Microsoft Edge (23)		0%	224.1 MB	0 MB/s	0 Mbps
MSYS2 terminal		0.6%	1.5 MB	0 MB/s	0 Mbps
Notepad++		0%	9.8 MB	0 MB/s	0 Mbps
Task Manager		0.0%	57.3 MB	0 MB/s	0 Mbps
Terminal (3)		0%	23.1 MB	0 MB/s	0 Mbps
Windows Explorer		0.3%	164.5 MB	0 MB/s	0 Mbps

An image showing the mGBA emulator process being stopped in Windows task manager.

This will remedy any unexpected issues when playing with assembly programs on the emulator. Of course if you're running Mac OS or Linux you'll need to follow a slightly different process but I'll leave it up to you to determine the best way to stop a process in your development environment. Before we move on I wanted to quickly refresh the details of the test assembly programs. Their location and directory structure as well as the best way to run them. All example assembly programs are located in the `cfg` directory directly inside the GenAsm NetBeans project folder.

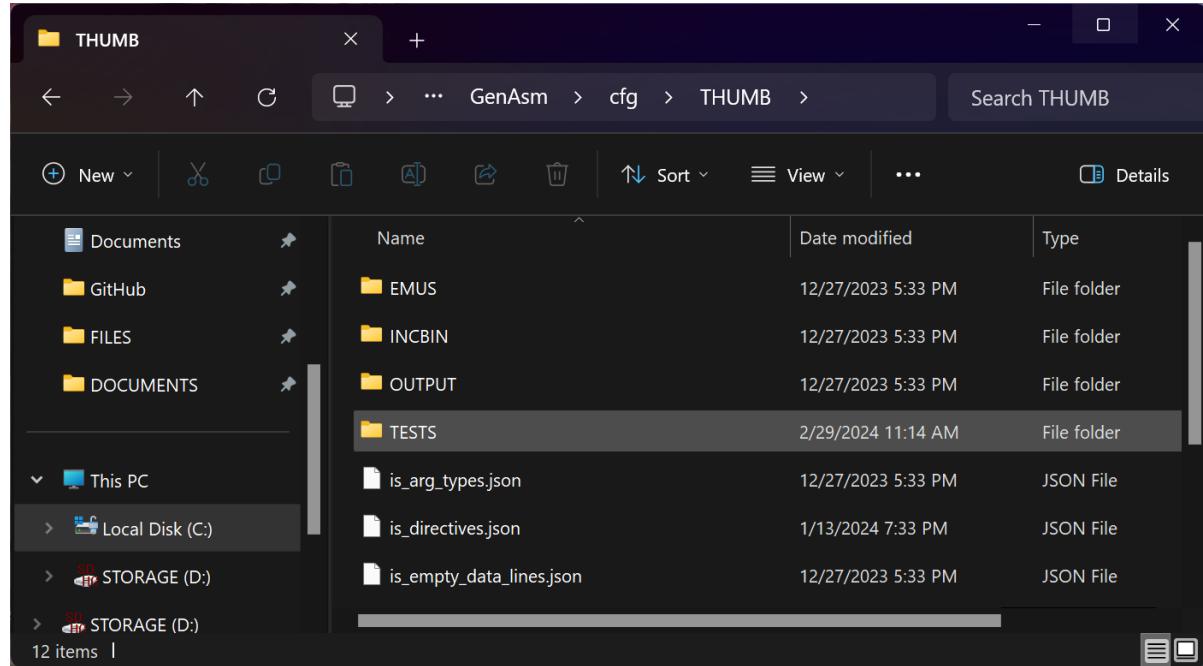
*Image 21-3. The cfg Folder Inside the GenAsm Project Folder*



An image showing the contents of the cfg folder inside the GenAsm project folder.

Because we're working with the ARM Thumb-1 instruction set the example programs are located in the THUMB folder. They are separated into two separate folders. The first, TESTS, contains the GenAsm assembly source code, a listing file, and an associated binary. The listing file and the binary file are from a test run of the program. There is also a unit test file called `vasm_answers.json`. The second folder is the OUTPUT folder. Each example program has a name and associated letter, A through O, with the source and output being in sub-folders of the same name respectively.

*Image 21-4. The THUMB Folder Inside the GenAsm Project's cfg Folder*



An image showing the contents of the THUMB folder inside the GenAsm project's config folder.

To assemble a test program simply change the name of the default test program on line 141 of the `GenAsm.java` file.

```
String targetProgram = "TEST_B_16BitXfer";
```

Then run the assembler by right-clicking on the `GenAsm.java` class in the NetBeans IDE and selecting 'Run File' from the context menu or by selecting 'Run Project' from the NetBeans menu. This will assemble the example program and write all resulting files to the `OUTPUT` folder with the same name as your target program. The GameBoy Advance emulator will run the little-endian binary output and a batch file, `run.bat`, is provided for convenience. Mac and Linux users will have to run the assembled program from a terminal window or port the simple batch script to a shell script for their development environment.

The last thing I wanted to discuss in this section are the unit tests. You can run them from the NetBeans IDE by right-clicking on the `TestProgramSuite.java` file and selecting 'Run File' from the context

menu. The unit tests are designed to re-assemble each example program and run a comparison with the binary output of that process to the `vasm_answers.json` file, included in each example program in the `TESTS` folder. As indicated in the naming of the answers file the values checked came from a different assembler. I suppose it wouldn't be good of a unit test if we only checked our own output and didn't compare it to another source of truth. Because there is a lot of assembly necessary to run the programs on the GameBoy Advance emulator the tests focus on checking pertinent work and data entries and ensure that the binary values match. Or, in the case with labels and data, that they match with a known offset.

That's all I wanted to mention about the test programs and their configuration. In the next section we'll take a look at a skeleton of the assembly program that is used to create example programs that run on the emulator.

## Example Programs: General Structure

All of the executable programs shown in the previous listing share a similar structure. In this section we'll review the general structure of an executable example program by discussing the `16BitXfer` program. Remember both the test and output folders are named, `TEST_B_16BitXfer`. We won't go into much detail about programming in ARM Thumb-1 assembly, the programming model, or the code necessary to get it to run in the emulator. That would be stepping on the toes of Mr. Akuma. You can learn all of that and much more directly from him. I'll list the links again here.

Keith Akuma

<https://www.chibiakumas.com/>  
<https://www.assemblytutorial.com/>

I should mention that I'll use the word procedure, for lack of a better word, to describe the equivalent of a function call in higher-level programming languages. I'll also refer to static label values, and set, labeled, memory locations as variables. Similarly, whenever a label is set to the memory location of a variable I'll refer to this as a pointer.

*Listing 21-2. General Structure of Executable Example Assembly Programs*

```
01 @TTL |16BitXfer|
02 @AREA |Program| @CODE, @READWRITE
03 @ENTRY
04
05 ;;;;;;;;;;;;;;;;;;;
06 ;ORIGINAL VASM CODE BY Keith Akuma.
07 ;https://www.chibiakumas.com/
08 ;https://www.assemblytutorial.com/
09 ;;;;;;;;;;;;;;;;;;;
10
11 ...
12
13     BL    `do_work
14     BEQ   `print_passed_address
15
16 print_failed_address
17     LDR   R1, [PC, `test_failed_address]
18     BL    `print_string
19     B     `inf_loop
20
21 print_passed_address
22     LDR   R1, [PC, `test_passed_address]
23     BL    `print_string
24     B     `inf_loop
25
26 ...
27
28 do_work
29     PUSH  {R0-R6, LR}
30
31     LDR   R0, [PC, `result_address]
32     LDR   R1, [PC, `value_address]
33     LDR   R2, [R1, #0]
34     STR   R2, [R0, #0]
35
36     LDR   R3, [PC, `answer_address]
37     LDR   R4, [R3, #0]
38     MOV   R8, R4
39     CMP   R2, R8
40
41     POP   {R0-R6, PC}
```

```
42
43 ...
44
45 ;Note this data block varies and is not aligned.
46 ;Care must be taken to ensure it starts on
47 ;a memory location divisible by 2 or 4
48 ;depending on what you are doing.
49
50           @DCHW    0x0000
51 value_address    @DCWBS   =value
52           @DCWBF   =value
53 result_address   @DCWBS   =result
54           @DCWBF   =result
55 answer_address   @DCWBS   =answer
56           @DCWBF   =answer
57 value          @DCHW    0x0000
58           @DCHW    0x23C1
59 result         @DCHW    0x0000
60           @DCHW    0x0000
61 answer          @DCHW    0x0000
62           @DCHW    0x23C1
63
64 ;;;;;;;;;;;;;;;
65
66 @END
67
68 @AREA |Data| @DATA, @READWRITE
69 @ENTRY
70
71 ;;;;;;;;;;;;;;;
72
73 ...
74
75 test_passed_address  @DCWBS   =test_passed
76           @DCWBF   =test_passed
77
78 test_passed
79           ;$FLIPSTRING |TEST PASSED|
80
81 test_failed_address  @DCWBS   =test_failed
82           @DCWBF   =test_failed
83
84 test_failed
```

```
85 ;$FLIPSTRING |TEST FAILED|
86
87 ;;;;;;;;;;;;;;;;;
88
89 @END
```

A skeleton of the TEST\_B\_16BitXfer program demonstrating the general structure of all executable example assembly programs.

The first thing I'll point out about the assembly program template is that it includes both a code and data section. The data section is mainly used to hold data that drives the visual display of the emulator including a simple font for drawing text. We also include some data in the code section of the template. This data is used to drive the main code of the example programs. This code is contained in the do\_work procedure, lines 28 – 41. The variables used by the procedure can be found on lines 57 – 62. You'll notice that the variable values are set using data preprocessor directives.

Just prior to the variables section of code are the pointers. These are special variables that store the address of the ROM memory location that the variables are stored in. When we load up a variable we start by pulling its address from a pointer, lines 32 and 36. Then using that address we load the variable's value into a register for use, lines 33 and 37. At the end of the procedure we do a comparison. In order to do this we need to move one of the values from a lower register to a higher register, line 38. The last operation performed by the procedure is to do a comparison of the values stored in the two registers on line 39.

Note that this example code is a simple 16 bit transfer, meaning that it loads up a location to store a value in. Then it loads up the value to store. Lastly, it then stores the given value in that location. Before the procedure returns the register that was used to hold the stored value is compared to a known value, an answer. This leaves the CPSR flags, condition codes, in the correct state so that we can branch based on the result of the comparison. This takes place after the call to the procedure on line 14.

If the values compared were equal then the BEQ opcode will fire and the print\_passed\_address procedure will execute, line 14. If the values are not equal the next section of assembly code to execute is the

`print_failed_address` procedure. Note that both procedures effectively stop the program by going into an infinite loop. You have to keep in mind that the ARM processor is always doing something. So in order to tell it to stop you have to give it an infinite loop or else the program will continue on and potentially crash.

Take a moment to look over the data section entries that define the text that is displayed on the screen when a test passes or fails, lines 75 – 85. Keep in mind that pointers point to a certain ROM memory location. That means the value they store is a valid memory address of a variable. That's all there is to it really. All the executable example assembly programs use this basic template. Take some time to look over and understand the assembly code in each one and certainly take a look at Mr. Akuma's websites for more information on the nuances of programming the GameBoy Advance hardware/emulator.

## Chapter Conclusion

That brings us to the conclusion of the example program review. In this chapter we talked about the setup of the example programs including how to quickly assemble one from the NetBeans IDE. We also discussed how to run the output of the assembly process in the emulator and what to do if something goes wrong and your emulator crashes.

We also took a detailed look at the general layout of an executable example assembly program. Make sure to take a moment to review the listing output file as well as the assembly source code for all of the included example programs. The listing file holds a lot of useful information and proficiency working with it is a must when working with any assembler, including ours.

## Chapter 22: Conclusion

Every journey has to come to an end, and this is that ending. I originally set out to write this book because I couldn't find a good answer to the questions I encountered while building an assembler. Coming from a background of programming high-level languages, the pretty ones like Java and C#, I had a bit of a tough time wrapping my head around assembly programming. My solution to this, of course, was to write an assembler. What better way to learn an assembly language?

After a few failed attempts the picture became clearer to me but so did the hurdles. First of all, building even a simple assembler is a fairly large undertaking. Not only do you have to build an assembler, but you have to assemble something, so you have to support an instruction set of some kind. This, in-and-of-itself presents numerous challenges. My solution was to create as simple as possible, but no simpler, an assembler and support the most concise but relevant instruction set I could find.

After some amount of deliberation and possibly due more to desperation than anything else I settled on the ARM Thumb-1 due to its small but effective list of supported opcodes. The 16-bit instruction width also didn't hurt. Less bits means simpler opcode arguments. Working slowly from the JSON data structure necessary to describe the instruction set I create the

classes you've reviewed and built an assembler that's easy to extend, fairly robust, and can be thoroughly reviewed in less than 700 pages, give or take.

Faced with another daunting task, how do I explain all of this so someone can learn it and build something better, faster. I hope you liked my solution to this problem. Given that a program of this complexity can't be developed as a series of simple steps I chose to review in detail each class in the project, within reason of course, and literally to review each class field and method as a means of developing an understanding of what a class does and how it is used.

## Knowledge Summary

We've covered a tremendous amount of material in this text. No small feat. Let's take a moment to review some of the main points of the material we've covered in each chapter of this text.

- **Chapter 1: Introduction** – A quick start chapter that had us editing assembly source code in the first few minutes of the text. In this chapter we setup our development environment, assembled a test program and ran it on an emulator displaying a unique name on the emulator's screen.
- **Chapter 2: Introduction to Assembly Programming** – In the second chapter of this text we jumped right into another crash course. This time we introduced some basic concepts of assembly programming. We also reviewed, at a high level, the listing output file. This was all intended to give you as much understanding about assembly programming as little time as possible. Lastly, in this chapter we outlined a number of concepts that, although out of the scope of this text, are important to have some knowledge of.
- **Chapter 3: ARM Thumb-1 OpCodes Part 1** – The first part of a detailed review of the ARM Thumb-1 instruction set with examples of the JSON data file objects used to define the instruction set's opcodes and opcode arguments.

- **Chapter 4: ARM Thumb-1 OpCodes Part 2** – The second part of a detailed review of the ARM Thumb-1 instruction set with examples where applicable.
- **Chapter 5: Preprocessor Directives** – In this chapter we went over all of the preprocessor directives supported by the GenAsm assembler. In each case we reviewed the syntax of each directive and provided demonstrations of the different types of directives in use.
- **Chapter 6: Standard Directives** – A detailed review of the standard directives supported by the GenAsm assembler with demonstration code and sample listings.
- **Chapter 7: Instruction Set Data Files - ARM Thumb-1** – A review of the JSON data files used to describe the Thumb-1 instruction set and supporting objects and data.
- **Chapter 8: Simple, Extensible Coding Model** – An introduction to the structure of the project from the perspective of the Java class structure. Including a detailed review of an instruction set implementation and how to add a new file to an implementation.
- **Chapter 9: The Utility and Helper Classes** – A detailed review of the cross-cutting utility and support classes used by the GenAsm project.
- **Chapter 10: The Base Classes Part 1** – The first part of an in-depth review of the project's base interfaces and classes.
- **Chapter 11: The Base Classes Part 2** – The second part of an in-depth review of the project's base interfaces and classes.
- **Chapter 12: The Static Main Entry Point** – In this chapter we documented the program's command line arguments and different calling patterns. We also reviewed the hard-coded argument approach that's more efficient running the assembler directly from the IDE.
- **Chapter 13: The Holder, Loader, and Exception Classes** – In this chapter we got to see the project's base classes in action a second time as we reviewed some the project's holder, loader, and exception

classes.

- **Chapter 14: The Preprocessor** – In this chapter we’re introduced to the preprocessor and took a look at the Java class that powers the assembler’s preprocessor.
- **Chapter 15: The Lexer** – In this chapter we’re introduced to the class responsible for lexerizing the input text and separating it into pieces of text.
- **Chapter 16: The Tokenizer** – The next step in the assembly process introduces us to a data driven tokenizer. This class is responsible for identifying each artifact generated by the lexer.
- **Chapter 17: The Assembler Part 1** – The first part of the assembler review we take a look at the class’ fields and support methods.
- **Chapter 18: The Assembler Part 2** – The second part of the assembler review where we cover steps 1 – 10 that the assembler takes to prepare the data structure created from parsing the assembly source code.
- **Chapter 19: The Assembler Part 3** – The final part of the assembler review where we covered steps 11 – 13 of the assembly process and got to see how the binary representation of each line is generated.
- **Chapter 20: The Linker** – The last class review in the text where we took a close look at a simple linked that’s responsible for connecting up to two areas together into one direct binary file.
- **Chapter 21: The Test Programs** – This chapter is all about assembly programming. We got to see the general structure used by the executable example programs. We also reviewed four such programs in detail going over each line of assembly in the operative procedure.

Throughout the course of the text, you’ve also looked at countless lines of code, all manner of Java interfaces, classes, enumerations, class fields and methods, and a number of demonstrations.

# Special Thanks

I'd like to take a moment to thank my beautiful wife Katia for putting up with me while I code and write during the bachelor. I'd also like to thank her for her help with the charts and cover art on this book. Thank you, Elkus!

Katia Pouleva  
<https://katiapouleva.com>

I'd like to give special thanks to Keith Akuma for sending me an ARM Thumb-1 implementation of his GameBoy Advance screen demonstration source code. Without it I wouldn't have been able to port it to the GenAsm source code syntax and complete the executable example programs for this text. I might not have even finished the text if it wasn't for Mr. Akuma. Thank you very much sir!

Keith Akuma  
<https://www.chibiakumas.com/>  
<https://www.assemblytutorial.com/>

I'd also like to take a moment to note all the little online tools I used at various stages in the development of the software and the text. The URL for each tool used can be found in the `cfg` folder of the GenAsm project in the `project_links.txt` file. Last but not least I'd like to thank ARM for their documentation and the availability they provide in accessing, and also for designing such an efficient instruction set in the ARM Thumb-1 ISA.

## Where to go From Here

You've just finished reading this text and you're wondering where do you go from here? Here are a few ideas:

- **Implement the ARM 32-Bit Instruction Set:** Many of the opcodes are similar and share a similar argument structure. With a little bit of hard work and a couple of late night you should be able to expand the use of

this software greatly.

- **Implement the RISC-V Instruction Set:** Create a set of JSON data files and associated classes to support the RISC-V instruction set.
- **Add Multiple File Support:** This software was designed to be simple and as such only supports one assembly source code file at a time. See if you can expand on this and carry multiple files through the assembly process and to the linker.
- **Port the Project:** Port this project over to C# or another to practice your multi-platform coding.

You could also spend a significant amount of time adding to the project's preprocessor or standard directives. Lastly, you could work on a few more demonstration programs to really hone your assembly programming skills.

## Saying Goodbye

This is the worst part of the book because it means saying farewell, but it also means I'm done writing. I sincerely hope you learned something from this text and that it has helped you in your endeavors. Ciao, ciao.