Carleton University

Department of Systems and Computer Engineering

SYSC 3006

Computer Organization

Assembler User Guide

Assembler: V6

Guide Version: 2.2

The Assembler is distributed as an html file (the assembler is written in Javascript). As a result, it will (should) run in any browser that supports Javascript (note: Javascript is not related to Java in any way). The Assembler was developed in Firefox using Firebug, and has also been validated to run (at least for a test program) in Internet Explorer and Opera.

Note: you must allow the page to run scripts … the assembler is just a script.

This Guide assumes you are using a Windows environment. If not, it is assumed that you have dealt with the differences before and the conversion of terminology will be simple ☺

Introduction

An assembler converts assembly language source code into an object format that can be loaded and executed in the SYSC 30006 Debugger circuit.

The assembly language syntax is exactly that discussed in class. Since the lecture slides already discuss the syntax, it is not repeated here (exception: a discussion of the DCD directive is given later). These notes discuss how to use the Assembler as a software tool.

The Assembler is written as a web page. This is good since it will run in any browser, but it also has a limitation. For general security reasons, web pages are not permitted to read and write to files. This limitation is discussed below in “Files and the Assembler”.

The Assembler can be loaded by opening the distributed file (SYSC3006Assembler-v-6.html) using the File → Open menu option. Many browsers have the Menu Bar disabled by default at installation, so you might not be able to see the File Menu. You can either enable the Menu Bar (right-click on the top-most bar of the browser’s window and select the Menu Bar), or use the Ctrl+O hotkey to start the File Open dialog (Ctrl+O = press O while holding the Ctrl key down).

When the assembler loads into the browser it will look something like Figure 1. Your screen may look different, depending on your default font and the size of the browser window. There are 4 sub-windows. The Source Code pane is where the assembly language source code is entered. To assemble the source code, press the Assemble button. If there are any assembly errors, they will be reported in the Error Message pane. The assembler stops as soon as it finds the first error. Fix that error, and then press Assemble again. Repeat until there are no more errors. When no errors are encountered by the

Assembler, it will report “NO ERRORS” in the Error Message pane, present an assembled version of the code in the Assembled Code pane, and present a Logisim compatible version of the code in the Object Code pane.

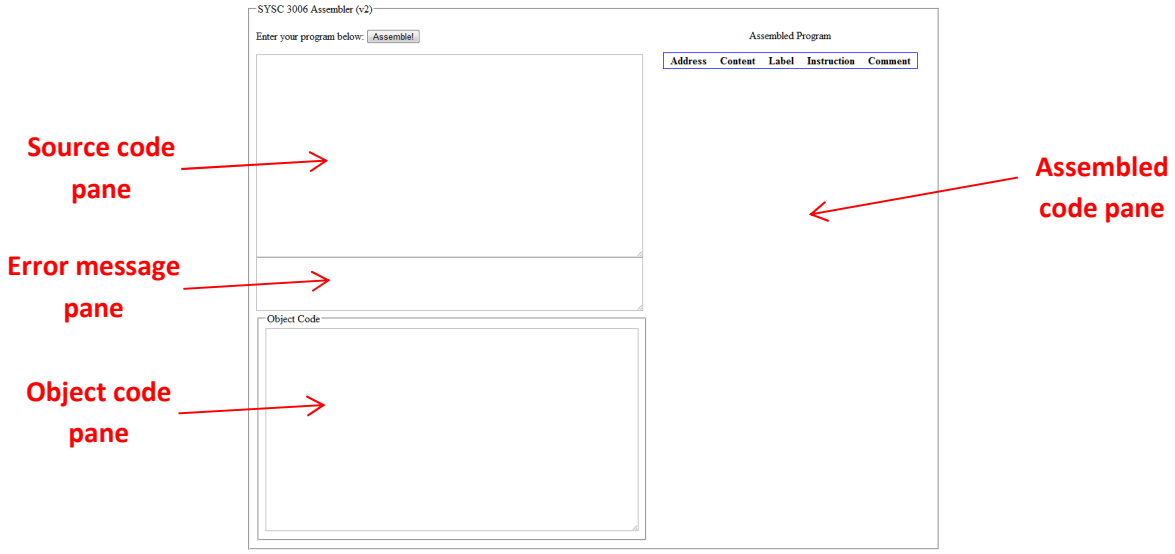

Figure 1 Assembler Open in Browser

An assembly language source code file called TestProgram SRC is available on the course website and will be used as an example in this document. The file is a simple text file and contains the following program:

```
        B Go                ; unconditional Branch over the Minus1 variable
Minus1 DCD #-1              ; a variable named Minus1 and containing the value “-1”
Go                          ; the target label declaration for the B instruction
        LDR R0, [ Minus1 ]  ; R0 = -1
        DCD #0xFFFFFFFF     ; breakpoint instruction is encoded as 0xFFFFFFFF
```

To load the program into the assembler, it must be first opened in a suitable text file editor. Notepad is OK to use. Prof. Pearce uses Notepad++ (http://notepad-plus-plus.org/) since it is free and has some very nice features that Notepad does not have. Notepad++ appears in some of the screen shots in this guide.

NOTE: Do not use a word processor like Microsoft Word when working with files associated with the assembler unless you have absolutely no choice. In general, word processors insert all kinds of control characters to help with formatting, and these control characters confuse the assembler.

Figure 2 shows the test program open in Notepad++.

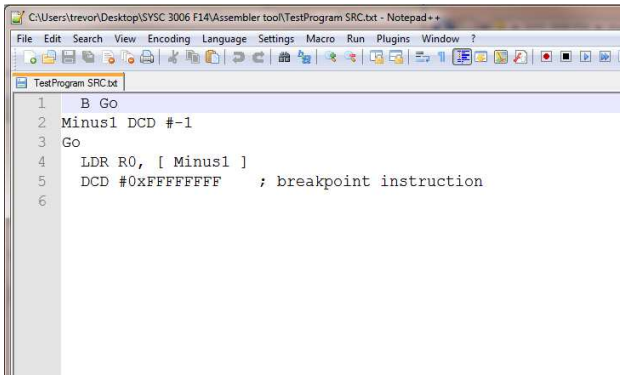

Figure 2 Test Program Open in Notepad++

To copy the code to the assembler:

1. Select all of the code. One way to accomplish this is to use Edit → Select All (also supported by Notepad)
2. Copy it to the Windows clipboard. One way to accomplish this is to use Edit → Copy (also supported by Notepad).
3. Right-click in the Assembler's source code pane, and select Paste

Figure 3 shows the test program loaded into the Assembler's source code pane.

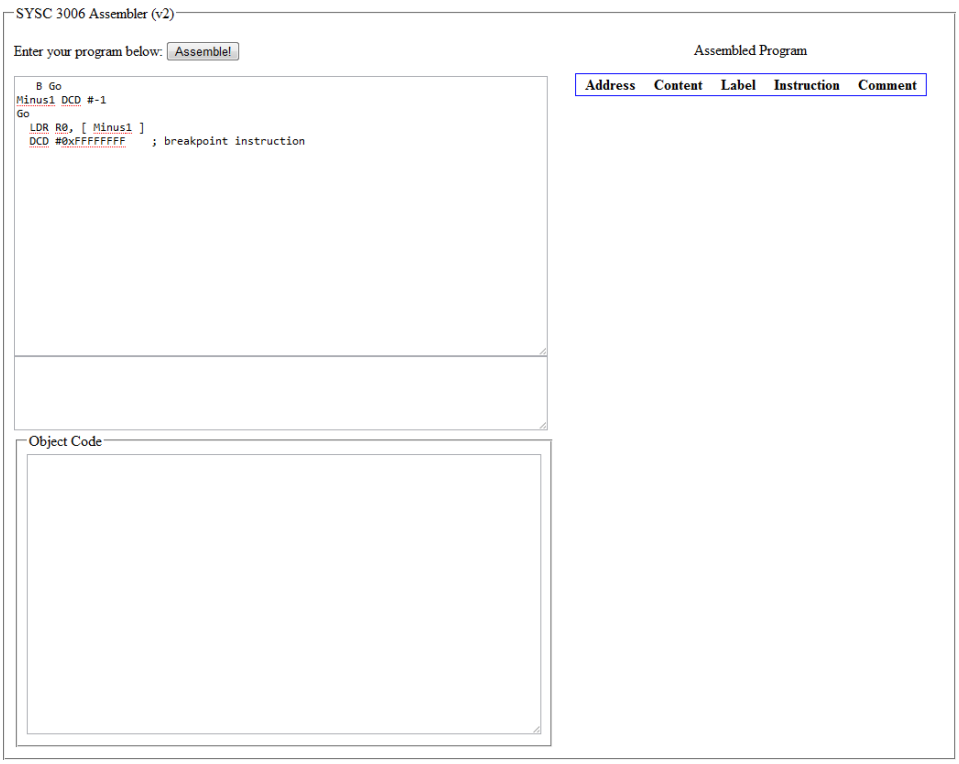

Figure 3 Test Program in Assembler Source Code Pane

Pressing the Assemble button will result in the scenario shown in Figure 4. Note: (1) no errors were encountered, (2) the assembled code is present together with the memory assigned for each element of the program and the encoding of each element, and (3) the Logisim compatible object code is present.

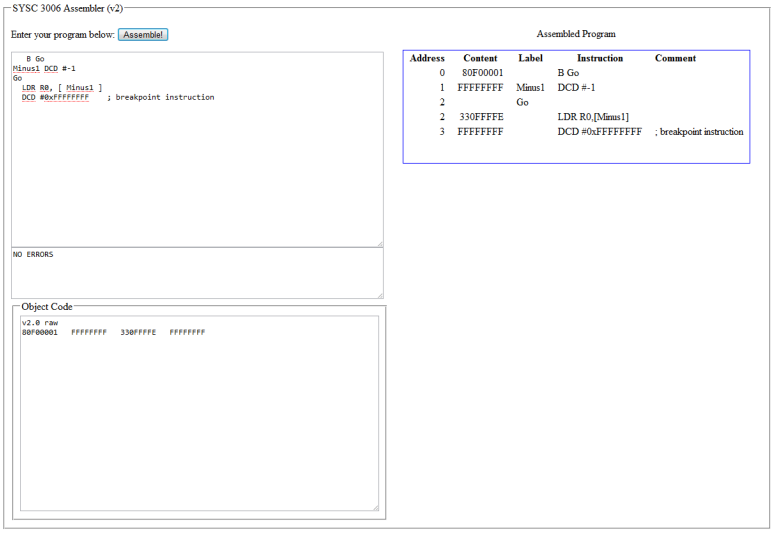

Figure 4 Assembled Test Program

Consider an error scenario. Suppose that the first instruction had given the Branch target as “GO” (all upper case) instead of the declared target “Go” (lower-case “o”). This is an error, since labels are case-sensitive. Figure 5 shows the resulting Assemble. The error Message pane shows the encountered error along with a (helpful?) error message. The error is described by its row and column (aside: Notepad++ clearly shows line numbers, but they start at 1 while this version of the Assembler starts line numbers at 0). The original line is also displayed and a “^” is shown beneath the line indicating the position in the line where the assembler encountered the error. No Assembled code or object code are given since the Assemble encountered errors.

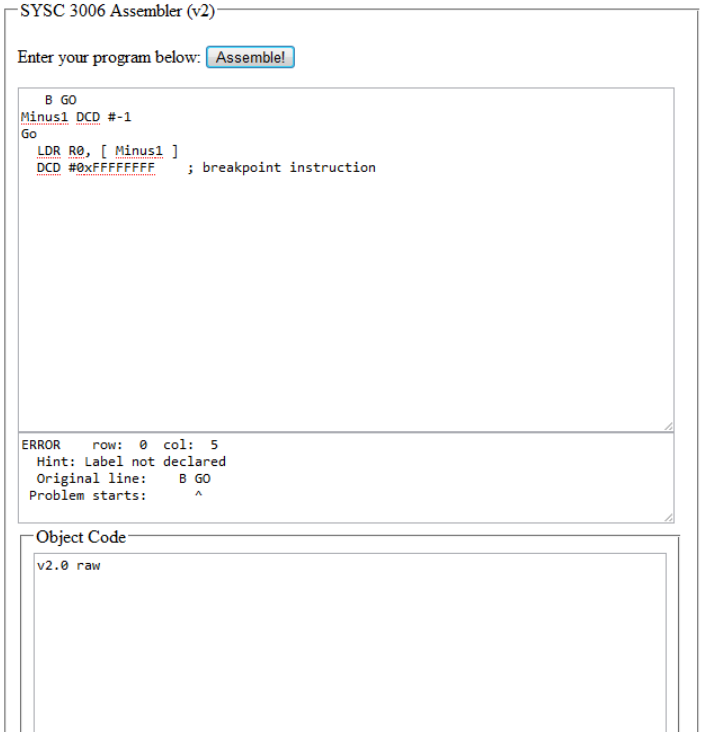

Figure 5 Error: Reference to GO

Once a program has been assembled cleanly (i.e. assembled without errors), the object code can be saved to a text file for loading into a Logisim Memory component. The first line of a Logisim memory image file must be "v2.0 raw", and this is placed in the Object Code pane by the Assembler. Saving a file can be accomplished using the following steps:

1. Select all of the text in the Object Code pane. One way to do this is to hold the left mouse button down while dragging it across the text (to-to-bottom, or vice versa). Another way is to click inside the pane to place the cursor in the pane and then use Edit → Select All.
2. Copy the object code text to the Windows clipboard. One way to do this is to right-click over the selected text and then select Copy. Another way is to use Edit → Copy.
3. Open a new text file in a text file editor (e.g. Notepad++), for example using File → New.
4. Paste the object code text into the new text file. One way to do this is to right-click over the selected text and then select Paste. Another way is to use Edit → Paste.
5. Save the text file with a useful name, e.g. Lab7ProgramOBJ.txt

The saved file can now be loaded as a memory image into the Debugger circuit. See the Debugger User Guide for details.

Figure 6 shows the Test Program object code after being pasted into a New Notepad++ file.

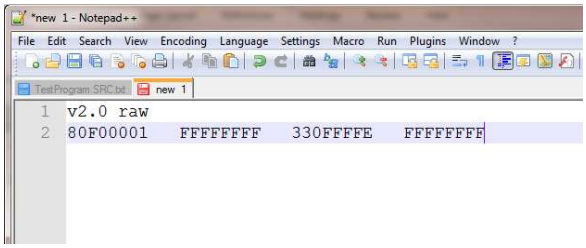

Figure 6 Object Code in a New Notepad++ File

Note that the Source Code pane is just a textarea and can be edited. Just click on the text area to place the cursor in the text and then edit the text. This allows a program to be loaded from a file, modified if necessary, and then assembled. If you do this, be sure to save the modified source code before closing the Assembler! The source code can be saved using the same steps used to save the object code (although it might be more appropriate to save the modified source code in the original source code file instead of a new file).

Files and the Assembler

Having the Assembler work in a browser is very convenient, but it can make the handling of files a bit awkward at first. Using an editor that allows multiple files to be open at once can simplify the use of the Assembler. Notepad++ allows multiple files to be open at once, while Notepad does not.

Here is one way to manage files during a development session while developing a program for use in the Debugger circuit:

1. Open the source file in an editor tab. Keep this tab open during the entire development session.

2. If you have a previous object code file for the program, open it in another editor tab. Keep this tab open during the entire development session. If you do not have previous object code for the program, open a new editor tab for the object code.
3. Copy the source file to the Assembler Source Code pane.
4. Modify the code in the Source Code pane until it assembles cleanly. At this point it is probably a good idea to copy the (modified) code from the Source Code pane back over to replace the original source code in the editor. (An experienced programmer would save the code under a new file name and keep a copy of the old code.)
5. Once the code assembles cleanly, copy the object code over any contents of the editor tab set up in step 2.
6. Save the object code file (for loading and debugging in the Debugger circuit), but leave the tab open for the next development iteration.
7. If at any point you wish to stop the session and keep your last set of modifications, copy the (modified) code from the Source Code pane back over to replace the original source code in the editor and save the file.

The first few times the copy/save of files must be done it can be awkward, but after a few times the process becomes quite easy and quick.

DCD Directive

The DCD (DeClare Data) directive is used to declare a memory location to be used as a variable. The general syntax for the directive is:

[Label] DCD [numeric-literal] [; comment]

Anything enclosed in [and] is optional. If present, the Label is the name of the variable. If present the numeric-literal defines the initial value that should be loaded into the variable's memory location when the program is loaded. As always, comments are optional.

Here are some examples:

```
        DCD             ; reserves an un-named, un-initialized word

X       DCD             ; reserves a memory word for an un-initialized variable named X

Y       DCD     #12     ; reserves a memory word for a variable named Y, initialized to 12 (decimal)

X       DCD     #0x10   ; reserves a memory word for a variable named Z, initialized to 10 (hex)
```

Note that decimal values for DCD numeric-literals may be specified as negative (using 2's complement encoding). For example

```
Minus1 DCD      #-1     ; reserves a memory word for a variable named Minus1, initialized to -1
                        ; the variable is initialized to the value 0xFFFFFFFF i.e. -1 under 2's complement
```