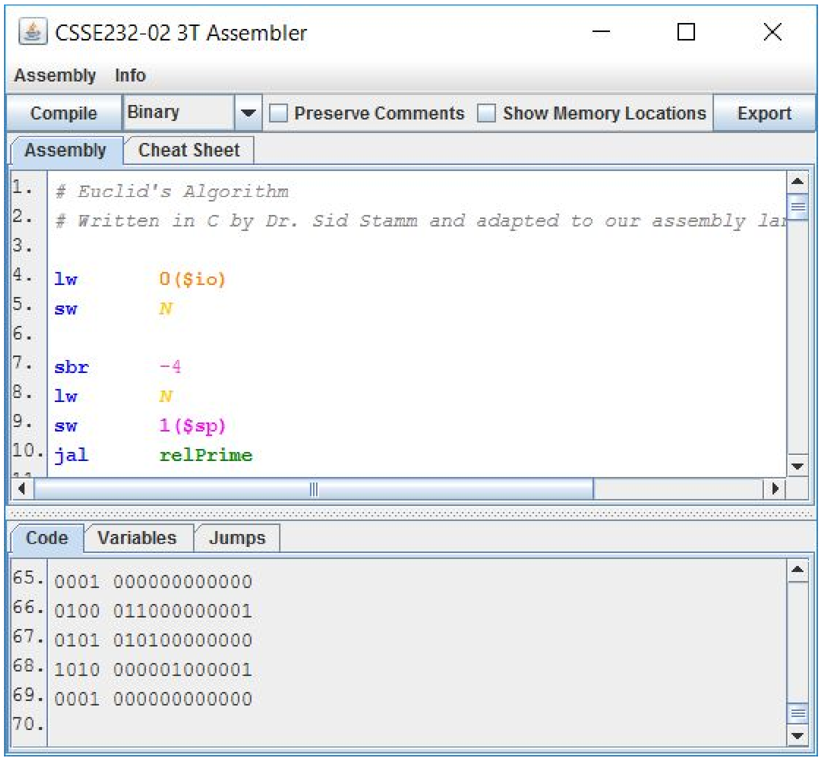


```
# Euclid's Algorithm
# Written in C by Dr. Sid Stamm and adapted to our assembly la

lw       0($io)
sw       N

sbr      -4
lw       N
sw       1($sp)
jal      relPrime
```

```
65. 0001 000000000000
66. 0100 011000000001
67. 0101 010100000000
68. 1010 000001000001
69. 0001 000000000000
70.
```

Team 3T Final Project: General Purpose Processor Java Assembler

CSSE 232 - Computer Architecture 1

Designers:

Calvin Weaver

Josh Osborne

Jason Duncan

Tyler Reinhardt

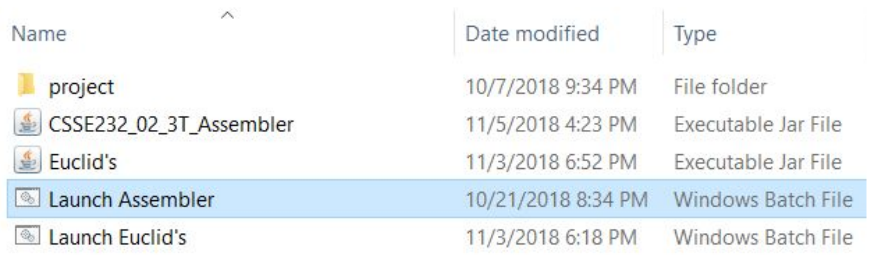
Table of Contents

Overview

To compliment our processor, our team designed a simple program assembler in Java. This assembler handles the translation of assembly and custom syntax elements into a memory initialization file (`.coe`) that can be read by Xilinx. The assembler features an interface for creating `.txt` files, loading existing `.txts`, and saving edited `.txts`. This is for the purpose of quickly writing, saving and editing custom instruction sequences for our processor. Consequently, all loaded text files are assumed to be of our custom syntax and highlighted/analyzed accordingly. The assembler also features a number of tools to analyze the translation process from `.txt` to `.coe`.

The assembler was compiled with Java 8, and thusly requires a version to be installed on the user's computer. The project itself was created using Eclipse Oxygen, and the contents can be located in the "project" directory.

The assembler can be run by double-clicking the `CSSE232_02_3T_Assembler.jar`, or to optionally run the assembler with the command prompt visible (for debugging purposes) double click `Launch Assembler.bat`. Launching the assembler with the `.bat` has no effect on the assembler's functionality other than showing the command prompt.



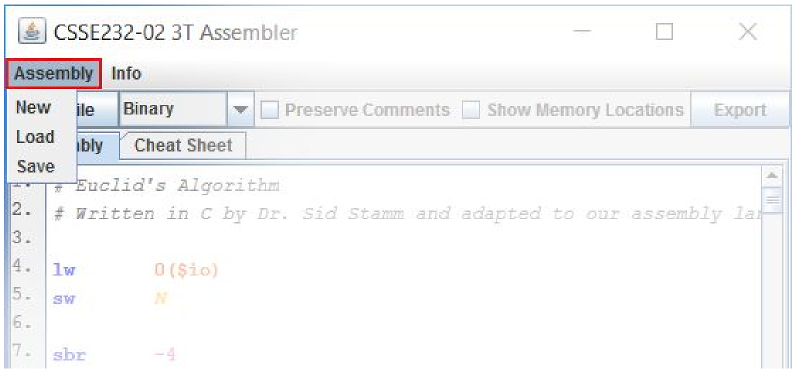
Name	Date modified	Type
project	10/7/2018 9:34 PM	File folder
CSSE232_02_3T_Assembler	11/5/2018 4:23 PM	Executable Jar File
Euclid's	11/3/2018 6:52 PM	Executable Jar File
Launch Assembler	10/21/2018 8:34 PM	Windows Batch File
Launch Euclid's	11/3/2018 6:18 PM	Windows Batch File

Managing Files

All files edited through the assembler are assumed to be composed solely of elements that match the syntax of our language. As development started with `.txt` files, the assembler was made to handle that file type.

Creating a new .txt file:

1. Click on the "Assembly" drop-down menu in the upper-left to bring up the file operations menu.

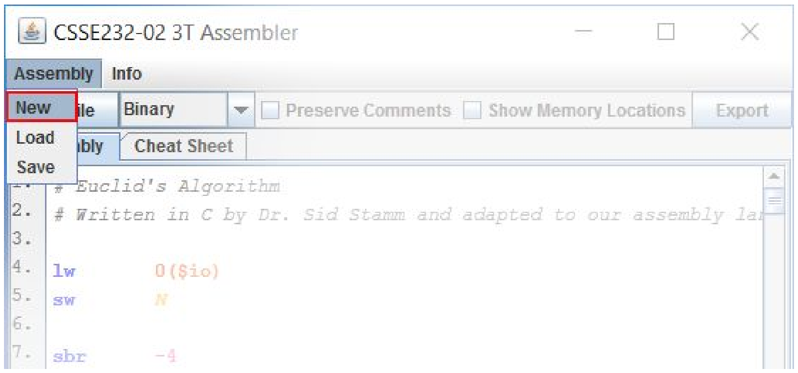


```
# Euclid's Algorithm
# Written in C by Dr. Sid Stamm and adapted to our assembly la

lw      0($io)
sw      N

sbr     -4
```

2. Click on the "New" button in the "Assembly" drop-down menu.
   (Optionally) The assembler will ask you to save existing work if there is a file already open.



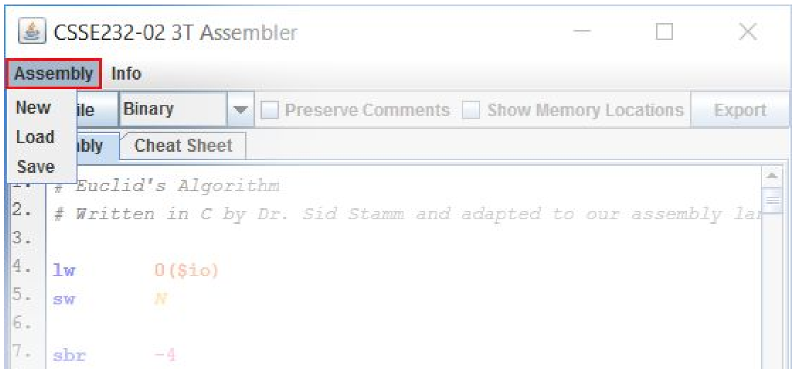
```
# Euclid's Algorithm
# Written in C by Dr. Sid Stamm and adapted to our assembly la

lw      0($io)
sw      N

sbr     -4
```

Loading an existing .txt file:

1. Click on the "Assembly" drop-down menu in the upper-left to bring up the file operations menu.

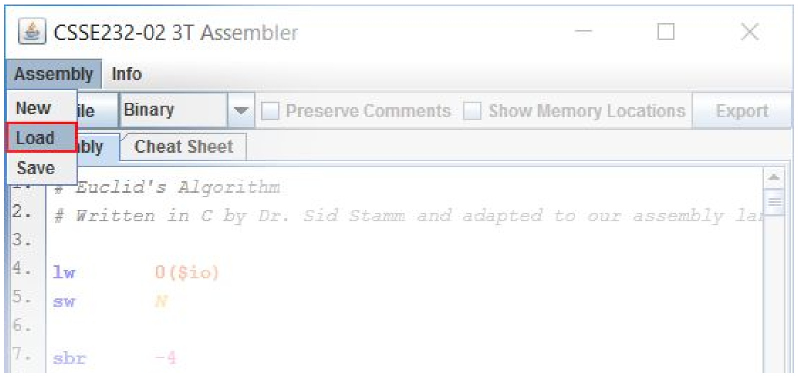


```
# Euclid's Algorithm
# Written in C by Dr. Sid Stamm and adapted to our assembly la

lw        0($io)
sw        N

sbr       -4
```

2. Click on the "Load" button in the "Assembly" drop-down menu. Then navigate to the location of the desired file.
   (Optionally) The assembler will ask you to save existing work if there is a file already open.



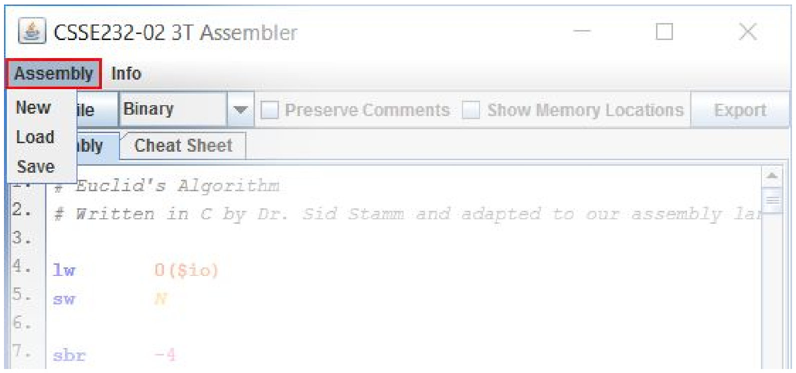
```
# Euclid's Algorithm
# Written in C by Dr. Sid Stamm and adapted to our assembly la

lw        0($io)
sw        N

sbr       -4
```

Saving an edited .txt file:

1. Click on the "Assembly" drop-down menu in the upper-left to bring up the file operations menu.

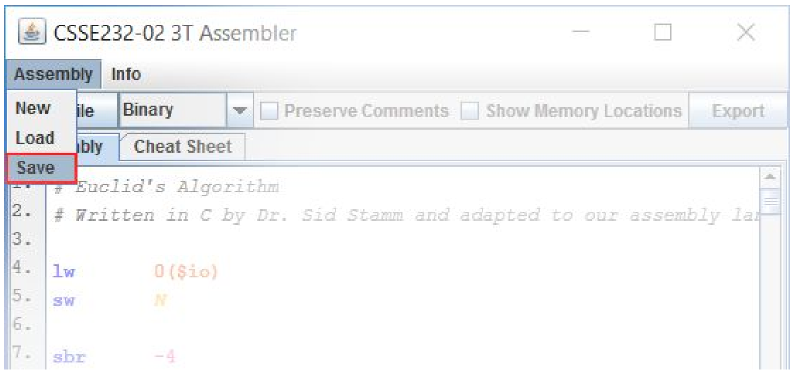


```
# Euclid's Algorithm
# Written in C by Dr. Sid Stamm and adapted to our assembly la

lw        0($io)
sw        N

sbr       -4
```

2. Click on the "Save" button in the "Assembly" drop-down menu. Then navigate to the location of the desired file.



```
# Euclid's Algorithm
# Written in C by Dr. Sid Stamm and adapted to our assembly lan

lw       0($io)
sw       N

sbr      -4
```

Reviewing Programs

Many tools are included to help users identify mistakes or syntax errors in their programs. Most importantly, the assembler provides real-time syntax highlighting. Each element type includes its own syntax coloring. The colorings are as follows:

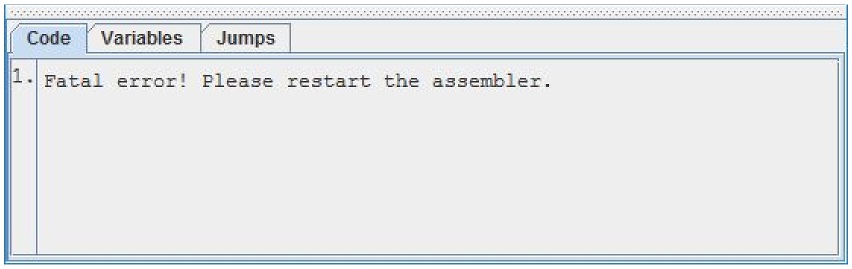
operations are blue	E.g. "**`sbr`**", "**`li`**"
comments are gray	E.g. "*`# This is a comment!`*"
values are pink	E.g. "`0xABCD`", "`0b0101`", "`1337`"
variables are orange	E.g. "***`VAR`***"
jumps are green	E.g. "**`RUN:`**"
stack pointer offsets are magenta	E.g. "**`2($sp)`**"
I/O offsets are orange	E.g. "**`2($io)`**"

Syntax errors are displayed with a bright red background. Both jumps and jump references follow the same highlighting rules as each other.

The assembler supports hex, binary and decimal formats as values. These values follow Java's notation standards ("`0x123F`", "`0b1100`", "`420`").

KNOWN BUG: **The assembler will not handle value overflows. If a value does not fit inside the 11/12-bit instruction space, the assembler WILL compile and WILL NOT produce an error. In this case, the compiled `.coe` will fail to load into Xilinx.**

With a few known exceptions, most errors are caught by the assembler and handled with a "soft-crash" state. The assembler will lock its text window and display a message in the compile output tabs.

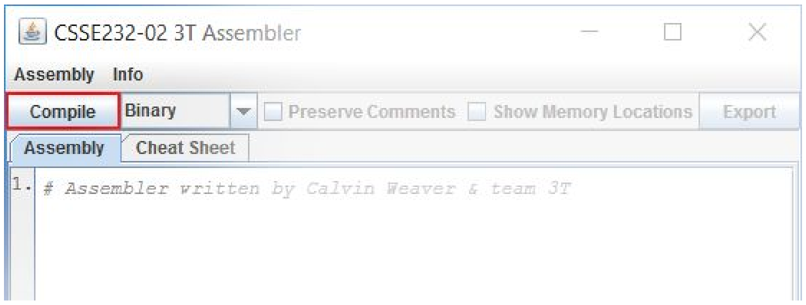


```
Fatal error! Please restart the assembler.
```

KNOWN BUG: **Entering an operation on the same line as a jump will cause a soft crash.**

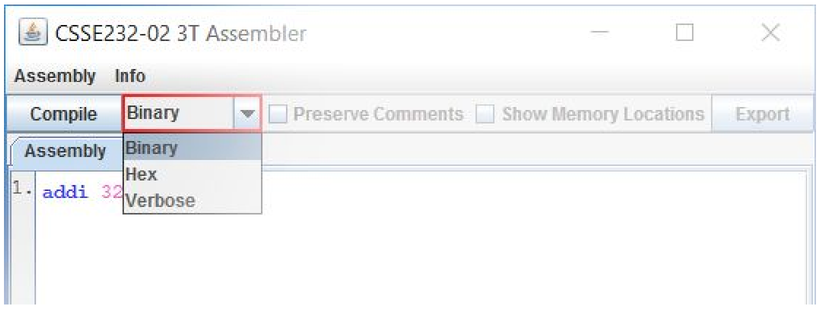
Compiling Programs

To compile the current program, click on the “Compile” button. This will translate the current program into machine code and display the translated list of instructions in the “Code” tab at the bottom of the window. Compiling the program will also create a list of all global variables that exist in the assembly program and write the list to the “Variables” tab. A list of all existing jumps will also be created and written to the “Jumps” tab. The assembler will not allow the user to begin a compilation if syntax errors exist in the current program.



```
# Assembler written by Calvin Weaver & team 3T
```

To help with the understanding of the compile process, many different compile output formats are available. These formats are exclusive to the “Code” tab. Formats are selected with the drop-down combo box immediately to the right of the “Compile” button.

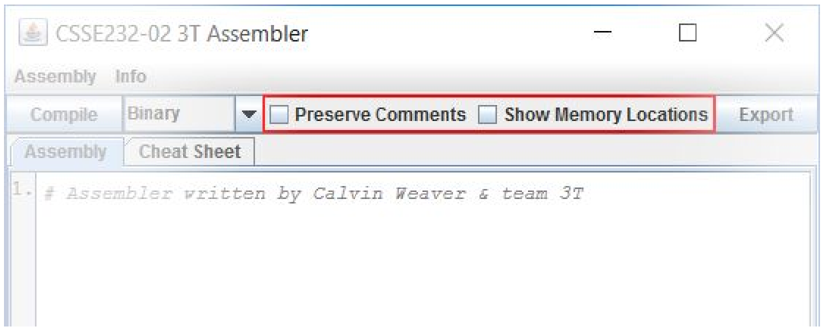


```
addi 32
```

The compiled “Code” tab outputs for the instruction “`addi 15`” are listed in the table below. The “Verbose” option exists for the purpose of debugging the assembler and identifying instruction memory locations.

Format	Output
`Binary`	`0001 000000001111`
`Hex`	`100f`
`Verbose`	`addi[15]`

Two checkboxes exist on the assembler's toolbar. Both are used to alter compile output formats. The first, "Preserve Comments", lists comments in their respective order of occurrence (between instructions). This is useful for identifying the location of instructions relative to the the rest of the program. The second, "Show Memory Locations", displays hexadecimal memory locations next to each instruction. These are the memory locations that each instruction will be created at during the export process. This is useful for gauging program length and debugging instructions during processor execution.

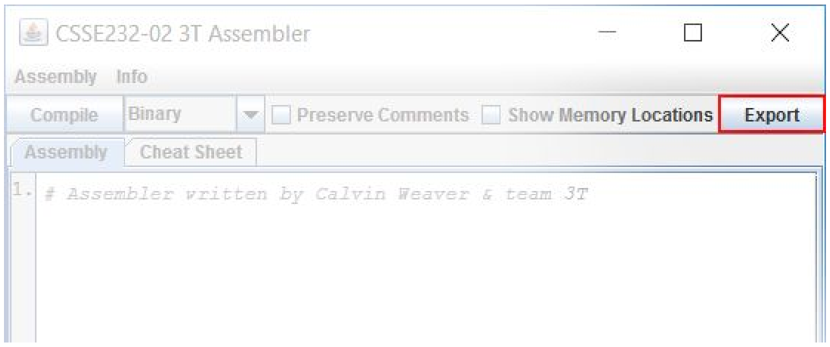


```
# Assembler written by Calvin Weaver & team 3T
```

KNOWN BUG: **If the current program contains the instruction "`freedom`", the program is disregarded and the entire Declaration of Independence is printed to the "Code" tab.**

Exporting Programs

To export the current program, click the "Export" button. A file explorer window will open to allow the user to select the destination folder. The current compile settings do not matter for exporting, as the assembler's export process will automatically overwrite them. Exporting will produce a single `.coe` Xilinx memory initialization file that contains the current program.



```
# Assembler written by Calvin Weaver & team 3T
```