How to use the compiler

This project compiles MIPS assembly code into machine code ready to be run a MIPS processor. The code is compiled from the asm.dat file, and must be contained in the same directory as the rest of the .asm files.

*How to compile correctly*

The resources needed to run the program correctly are provided. Every file (apart from the word docs) NEEDS to be in the same directory as Mars. The files that need to be with in the same folder as Mars are:

* asm.dat
* fileProcessor.asm
* lineProcessor.asm
* dataProcessor.asm
* Mars (obviously)

Two output files (dataSegment.dat, and instructions.dat) will be put here upon a successful run. These are the outputted code segments and data segments described below.

The program is run from fileProcessor.asm

*How to format code*

**Section 1: The Data Segment**

Every program made to work with this compiler must have a “.data” tag before any data in the source file. Even if your program doesn’t have any data, you must include the “.data” tag.

Every program made must also have a “.text” tag after all data and before all code. Even if there is no code in the “.text” segment, it is required for the compiler to differentiate between segments.

Every data entry needs a label except “.align”, which cannot have a label.

Data type supported and their descriptions/formats:

* “.align n” – This data type aligns the data in the data segment to whichever boundary is specified by the integer argument n. ( n = 0 for byte alignment, n = 1 for half-word align, n = 2 for word align, n = 3 for double word align ) The value is based on the first character in n, so if n = 15, the program would only use the character “1”.
* “.ascii str” – This data type moves into the data segment the string str and moves the pointer to the byte immediately after the end of the string. The string is to be formatted with quotes on both ends of the string. If these quotes are not there the compiler will terminate. Special characters supported are { “\n”, “\t”, “\r”, “\\”, “\”” }
* “.asciiz str” – This data type behaves exactly as .ascii, but instead places a null character “\0” after the end of the string and moves the pointer of the data segment to after the new character making the total length of the data segment entry the length of str + 1.
* “.byte b1, …, bn” – This data type takes the comma space delineated integer arguments and puts them into memory one after another. The arguments will be truncated if greater or lesser than that which a byte can contain. The arguments must be separated by exactly one comma and exactly one space. The last entry cannot have a comma after it.
* “.space n” – This data type will add to the data segment pointer n. This creates an arbitrary amount of blank space in the data segment.
* “.word w1, …, wn” – This data type behaves exactly as .byte, but instead of allocating memory byte by byte, it does so word by word. This function automatically aligns itself by word before storing data. The same rules for argument formatting from .byte applies here, use commas and spaces accordingly.

At the end of the program the data segment loaded at this stage will be put into a file called “dataSegment.dat”. This file contains in order all the words used by the data in the compiled program in hex. From left to right they are stored as words, so be aware that when reading it is formatted in little endian. The file would be loaded into address 0x10010000 by some linker program before runtime (not in the scope of this project).

**Section 2: Text Segment**

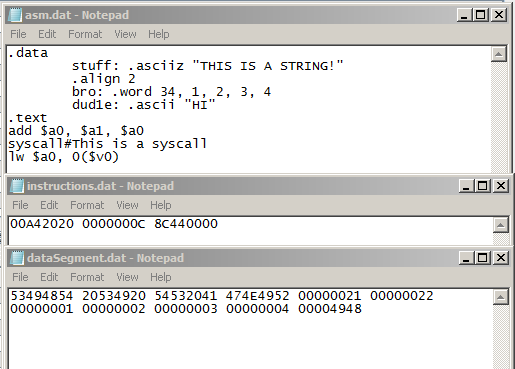
As stated above, every program made needs to have the “.text” tag to denote the start of the text segment.

The instructions you wish to compile must follow this set of guidelines:

* Instructions must have all arguments delineated by a single **space**.
* All references to registers use their letter names ($a0, $a1, $ra, $s2, etc.)
* Register zero (and only register zero) is the exception, it used exclusively $0.
* Blank lines are acceptable.
* Labels can go on a line before an instruction, or on a blank line alone.
* Multiple labels can point to the same address.
* Multiple labels can exist with the same name (the compiler will ignore duplicates).
* Comments can be at the end of lines.
* Comments can be on blank lines.
* Whitespace before an instruction (tabs, spaces, etc) is allowed.
* Syscall is supported.
* Due to no pseudo-instructions accessing labels from the data segment will not work.

All code compiled will be put into space delineated hex words in “instructions.dat”. This file will be generated if it doesn’t exist, or erased and written over if it does. This will be in the same directory as the source file. These instructions would be loaded into 0x00400000 by some linker, but that is beyond the scope of this project.

Only instructions from the CORE INSTRUCTION SET are supported.



Sample Code/Output