Connecting high and low: Adaption and use of tools to increase students’ understanding in programming

Mark Annala

Institutionen för informationsteknologi
Department of Information Technology
Abstract

Connecting high and low: Adaption and use of tools to increase students’ understanding in programming

Mark Annala

Learning assembly programming is often described as a major hardship for students in computing, making it harder to fully comprehend courses such as operating systems and computer architecture. This thesis explores the possibility to extend a tool for assembly programming in Mips, Mars, to better bridge the gap between assembly programming and high-level programming languages such as C. Two different approaches are explored, for Mars to execute assembly compiled by a web-compiler, and to execute C code intermixed with assembly in Mars. An extension of Mars was implemented with added functionality for executing one function in C and the possibility to, with some modifications, execute assembly code from a web-compiler. These extensions result in the possibility of Mars becoming an all-in-one tool used when learning assembly programming, if it is expanded with a cross-compiler for compiling C code into Mips assembly and with extra support for multiple tabs where both C code and corresponding assembly code can be seen next to each other.
Acknowledgements

I would like to direct a huge thank you to Professor Emeritus Pete Sanderson of Otterbein University for being so eager to help me with various questions regarding anything Mars-related. Without his help I would never have got going in the same manner that I did.

I would also like to thank my supervisor, Karl Marklund, for widening my scope and keeping me on track to finish in time and my subject reviewer, Aletta Nylén, for all the guidance when writing the report.
# Contents

1 Introduction 6  
1.1 Problem statement and scope 8

2 Background 10  
2.1 Different simulators and emulators 10

3 Method 14  
3.1 Understanding Mars 14  
3.2 Extending Mars 14

4 A deeper dive into Mars 16  
4.1 From startup to execution 16  
4.2 The Mars/Mips instruction set 18  
4.3 The magic of Mars’ system calls 18

5 Extending Mars to execute assembly code from third party 19  
5.1 Instructions not recognized by Mars 19  
5.2 Deciding on a calling convention 20

6 Extending Mars to execute C code 21  
6.1 An introduction to JNI 21  
6.2 Limitations 22  
6.3 Alterations made to the Mars source code 23  
6.4 Additions made to the Mars source code 23

7 Result 26  
7.1 Extending Mars to execute assembly code from third party 26  
7.2 Extending Mars to execute C code 27

8 Discussion 29  
8.1 Extending Mars to execute C code 29  
8.2 Extending Mars to execute assembly from third party 29  
8.3 Comparison of the two approaches 30  
8.4 Concrete examples 30

9 Conclusion 34

10 Future work 35  
10.1 Integrating a C to Mips 32 compiler in Mars 35  
10.2 Full implementation of C code functionality in Mars 35  
10.3 Changing register names of compiled assembly 35  
10.4 Automating manual work 36  
10.5 Survey on students learning 36
1. Introduction

Students who study computer science or information technology will come in contact with programming at varying levels of abstraction, sometimes writing code in assembly intended for one specific system architecture and other times writing code intended to run on multiple different machines. The problem many students face with the lowest abstraction level is that it is quite unintuitive as well as generally hard to grasp, which makes it difficult while attending courses in computer architecture or operating systems that require processor-oriented programming in assembly. That students have a difficult time grasping assembly programming is no novelty. Literature from 1985 witness of how students felt that the assembly part of their education was a 'major source of difficulty' [1]. More recent literature [2] from 2008 states both that assembly is thought to be a tough tool and that it requires a significant effort on comprehending the programming capabilities in order to reach to final projects implementation. Lastly, a study from 2013 [3] issued a survey to 30 students in their second or third year of studies about a new simulator and found that 90% of the participating students felt that programming in assembly is difficult.

One way to reduce the complexity of assembly programming could be to merge code on the lowest of abstraction levels with other parts, written in another language on a higher level that students are more familiar with, such as C. C has been around for a long time and is still highly regarded, but most importantly it is often used in education. For example, at Uppsala University it is used in courses such as Imperative and Object-Oriented Programming Methodology and Operating Systems and Process-Oriented Programming. This means that students are quite familiar with it, and it could therefore serve as a bridge to learn assembly. Another approach to increase students understanding could be to first translate C into assembly, and then step through the execution of that assembly code. Seeing how different data structures such as arrays and structs in C are represented in assembly, and also seeing how trivial functionality such as looping through an array is implemented on the lowest of abstraction layers could very well further students understanding.

Usually a simulator is used when studying assembly code, this because it is easier for the programmer to get a step-by-step overview of how the code is being executed. Using a simulator also comes with other advantages, such as back-stepping, GUI with point and click control and variable speed single-step execution. A popular simulator for the Mips architecture is Mars [4], which is used for several courses at Uppsala University. Mars works really well for education purposes, but the problems students face are still the same. Therefore, if Mars also could have the extra capability of being able to execute C code, or if students easily could compile their own C code into Mips assembly and then simulate it in Mars, many students would have a much easier time grasping the content that is taught.
Figure 1 shows the current Mars layout, where assembly code is written in the editor. Once the code is written it is possible to assemble the code by pressing the ‘wrench’ button and after that the code can be executed with the ‘play’ button. The vision of having Mars as an all-in-one tool for assembly programming can be seen in Figure 2. Compared to the current layout, it has (1) a tab for code written in C, (2) a tab for the corresponding code in assembly, and (3) a new button to compile the C source into Mips assembly.
1 INTRODUCTION

Figure 2: Mock vision of Mars with (1) a tab for code written in C, (2) a tab for the corresponding assembly and (3) a new button to compile the C source into assembly.

1.1. Problem statement and scope

Although the ultimate aim is to make Mars an even more powerful all-in-one learning tool for assembly programming, this work is a pre-study exploring paths towards that goal. In particular, this thesis investigates these research questions regarding the MIPS Assembler and Runtime Simulator, Mars:

- Is it possible to extend Mars with the added functionality of executing intermixed C and Mips assembly code?

- Is it possible to extend Mars to run Mips assembly generated by a third party compiler?

A prototype providing the suggested extensions has been implemented. This thesis describes the implementation together with observations of requirements for further
advancing this work.

All modifications made to the Mars source have been developed on a MacBook Pro running macOS Big Sur 11.2.3 and the version of Mars used for development is 4.5. All generated assembly code executed in Mars is generated by the web-compiler used in this work, Compiler Explorer.
2. Background

While studying operating systems or computer architecture students need to learn, at least to some degree, assembly. To make the material easier for students to take on, teachers often use some type of emulator or simulator which has capabilities such as a graphical representation of the processor registers, step-by-step execution and back stepping. The main difference between a simulator and an emulator is that a simulator only simulates a software environment in which the program runs. An emulator emulates hardware, and for the purpose of assembly programming a simulator is all that is needed.

Just using assembly code without a simulator makes it hard for students to really grasp how the code is executed, as the learning experience can be enhanced by 15-20% by using a graphical user interface [5]. Also, if no simulator (or emulator) is used, the assembly code which the students must write will depend on which hardware they have. This would force all students to use the same computer architecture (like in a computer lab), making it difficult for students to work from home. Simulators often give students the possibility to stop the execution at any time and inspecting what values are in each of the registers, something that is very beneficial for grasping how the code is executed.

2.1. Different simulators and emulators

Even though this work focuses on Mars and tries to extend it, it might be beneficial to have an overview of what other similar tools are available. Following is therefore a few tools which can be used for emulating Mips architectures or simulating Mips assembly code.

2.1.1. Mars

Mips Assembler and Runtime Simulator, or Mars for short, is a "lightweight interactive development environment (IDE) for programming in MIPS assembly language, intended for educational-level use" [4]. Mars was developed at the Missouri State University by Kenneth Vollmar and Pete Sanderson, and the current 4.5 version was released in August 2014. Mars is used for the assembling and execution of Mips assembly code and is very pedagogical with its single-stepping abilities and graphical visualization of the registers and their data. According to the authors, Mars’ greatest strength is its support for interactive debugging where the programmer can easily set breakpoints and step (even backstep) through the execution. Mars also has a few other tools which can be useful for understanding assembly programming, such as the Mips X-ray that shows how every
instruction is run through the processor hardware and a data cache simulator that shows cache data such as hit rate percentage.

### 2.1.2. Qemu

Qemu is much more than just a simulator of assembly code. It is capable of emulating whole operating systems with very good, almost native, performance by use of dynamic translation. It has a few available Mips architectures as well [6], namely:

1. A generic ISA PC-like machine "mips".
2. The MIPS Malta prototype board "malta".
3. An ACER Pica “pica61”.
4. MIPS emulator pseudo board “mipssim”.
5. A MIPS Magnum R4000 machine “magnum”.

Even though Qemu seems to be very powerful, it might not work very well as a tool for students to learn assembly programming. A study [3] from 2013 on a new simulator for the ARM architecture expressed that Qemu is meant for professionals and that over 90% of students who had tried to use Qemu had reservations because it is not friendly to novice users.

### 2.1.3. Hades

Hades is a Java-based visual simulation environment, made for creating and visualizing electronic circuits [7]. It has a lot of features that comes in handy for educational purposes such as editing circuits mid-simulation, pause and continue and point and click for the interactive switches in each circuit. However, since it is created for the visualization of circuits, it is on an even lower abstraction level than all the other tools mentioned in this section, and also lower than what this project is aimed at. Hades and its very low abstraction level would fit better for courses in computer architecture or courses on electrical circuits design. At a quick glance it is quite similar to Logisim [8], which is currently used in the course Computer Architecture at Uppsala university. If Hades were to be used it would be as a replacement or supplementation to Logisim, but for the purpose of helping students understand assembly programming, Hades does not help.
2.1.4. Spim

Spim [9] is another clear candidate when learning assembly programming. Spim and Mars are quite similar in many ways, but also quite different. Both of them have an intuitive GUI showing all the registers available and their contents, something that is beneficial from a pedagogical standpoint since the students can see how each register is updated upon single-stepping through the assembly code. One downside to Spim compared to Mars is that Spim does not provide a text editor in which you can write your assembly code, something that is available in Mars with the extra addition of syntax highlighting and tooltips with instruction specifications. When writing assembly code for the Spim simulator, the students must first write the code in a separate file and then load it into Spim. This process becomes more fluent in Mars where one first writes the code, then upon assembling sees how all the registers are filled with data. A study [10] by the authors of Mars did a student evaluation comparing Mars to Spim. They disclosed their involvement in Mars and asked the students to be objective in their survey. The survey introduced students to both Mars and Spim at the same time and found that students strongly preferred Mars to Spim, and most of the requested improvements to Mars where implemented in a later version.

Both Spim and Mars are quite old, the latest version of Spim dates back to January 2011 and Mars’s latest update came in August 2014. They both still work fine for educational purposes, but seeing as both tools are starting to come of age with no further development in the last seven to ten years, one could perhaps wish for a new tool better suited to increase students’ learning.

2.1.5. Compiler Explorer

The Compiler Explorer [11] is an online tool that can compile a multitude of different languages into an architectural specific assembly. The Compiler Explorers compilation from C to Mips GCC 5.4 is used in this work, as it is out of the scope to build GCC specifically for Mips. Compiler Explorer has an intuitive GUI which shows the high-level programming language (in the case of this thesis, C) on the left-hand side and the compiled assembly on the right. It has color coding which shows exactly how each program statement is translated, something that could be useful for students trying to learn assembly programming. The Compiler Explorer website uses GCC to cross-compile C source into Mips assembly. This means that the user can add additional arguments to the compiler, such as optimization flags (-03, -Ofast) and verbose flags such as -fverbose-asm. Using optimization flags can give students a really nice intuition on how compiled assembly with optimization differs from the regularly compiled assembly. The verbose flag help by adding comments to the code, such as which parameters are being saved into which registers etc. This can help with understanding the calling convention,
discussed further in Section 6.

2.1.6. Newer, browser-based simulators and emulators

Lately, there has been a lack of new additions to desktop based simulators. Instead there has been an increase in browser-based simulators. WeMIPS [12] and CPUlator [13] are two examples that both have similar interfaces and functionalities. CPUlator is more advanced since it supports three different architectures (two different Mips architectures but also ARM and Nios) and for each architecture, a few different systems. Migrating from a desktop environment to a browser does have some downsides though. For instance is automatic file savings to the hard drive unusual, something that is done automatically in for instance Mars every time a file is assembled.

Since no further development is being made to the desktop applications, browser-based tools may be the better option in the future if they keep being maintained and updated.
3. Method

To be able to examine whether it is possible to extend Mars with new functionality and later doing so requires a good understanding of Mars, and since Mars is written in Java, also the Java programming language. Complete understanding of all Mars’ modules is not needed but understanding how the code is parsed and later executed is crucial. To extend Mars, additional knowledge of the Java Native Interface (JNI) and creating shared libraries with GCC was also needed.

3.1. Understanding Mars

Mars consists of approximately 200 files containing a total of around 35k lines of code. Achieving any deep understanding of the code only by studying it was therefore not feasible within the scope of this work. Instead, one of the co-creators of Mars, Pete Sanderson, has shared his knowledge of the system and other Mars-related issues connected with the extensions strived for. Even though Mars is quite ‘hacky’ as described by Pete Sanderson, the initial work of understanding how the modules interacted with one another was rather straightforward. A major part of the codebase could be ignored since knowledge on how the GUI, text editor, threading etc. works were unnecessary as they needed no modification. The modules of interest that actually parsed, assembled and executed code could be located in a top-down manner.

3.2. Extending Mars

For this work, it was best suited to build the extensions aiming at extending Mars to execute C code in isolation. Not until the intended functionality was achieved, were they integrated into the main application. Even though integrating new modules into existing software is tedious, it was to be preferred over continuous development in this case. With continuous development, the time spent running the application to the point where alterations are executed would make the development process more time consuming.

The goal of extending Mars to execute assembly from third party had to be tested to ensure that Mars in fact could execute all assembly that the third party compiler generated. This was done manually by running the compiled programs in Mars, making it easy to adjust the programs if any errors were raised. Since the primary aim is for Mars to be able to handle code written by students learning assembly programming, the programs will be of relatively primitive nature. It was, however, important to make sure that the full functionality of C could be compiled into assembly and executed in Mars
and to satisfy this demand a large number of C programs of varying complexity were tested.
4. A deeper dive into Mars

When using Mars to execute assembly code, it is possible to either create a new file and write assembly code to it or to load an already existing file from the hard drive. Once the file to assemble is loaded the user will press the assemble button, which will assemble the file and load the proper segments. If the file is assembled without errors it can be executed, either one step at a time forward or backward, or continuous with variable speed execution. Before the execution starts it is possible to add breakpoints to the code, which will stop execution at any given point.

Although Mars is easy to use as an end-user, what is going on is quite a bit more complicated. Following is therefore a walkthrough of how Mars executes an assembly file internally.

4.1. From startup to execution

![Figure 3: Overview of how Mars’ modules interact when assembling and executing.](image)

The class responsible for starting Mars with the right arguments is the MarsLaunch class. It is not usual for students to run Mars from the command line without GUI, but for example teachers can do so to easily assemble and execute multiple Mips assembly source code files one after the other.

Thanks to the debug options, a command which when enabled prints debugging information to standard output, it is quite easy to see where in the source code the file processing begins. The `TOKENIZING BEGINS` is one of these printouts and the method that is run just after starts the tokenizing. An overview of how Mars assembles and executes an assembly file can be seen in the marsLaunch module in Figure 3. Figure 3 also gives an intuition on how Mars’ different modules interact, and how these modules are extended is described in Section 6.
4.1.1. Tokenizing and pre-processing

The prepareFilesForAssembly method in the MipsProgram class is the first method to run on one or more assembly programs. It starts by reading all assembly source code from a file into the internal structure of the MipsProgram object and then continues to tokenize it with help of the tokenizer class.

The Tokenizer uses pre-processing in the processIncludes method to expand any directives in the assembly program. When this method is called, all the assembly source code is line by line in an ArrayList of Strings. Since this is the very first step of processing, it is also the perfect place to make any extensions that needs the source code to be unaltered. See how these extensions are made in Section 6.

4.1.2. Assembling

Once the Tokenizer has finished it is time for the Assembler, which is also called from the MarsLaunch class. The Assembler class has a main method, assemble, which creates an ArrayList representing the assembled program where each member of the list is a ProgramStatement object. The ProgramStatement contains the source, the intermediate, and the machine binary representation of the program statement. The Assembler runs through the source three times: the first run is to verify syntax, generate symbol table and initialize data segment, the second pass generates basic assembly and the third pass generates machine code.

4.1.3. Execution and simulation

The MarsLaunch class then continues with initializing the program counter, usually to .main, and then starts to simulate the program.

For every instruction available in Mips, called BasicInstruction, there is an implementation in the InstructionSet class for how that instruction is supposed to be executed and simulated. The Simulator class, which takes care of the simulation, delegates the actual simulation work to the BasicInstruction class’s simulate method. The Simulator simulates every instruction that way and updates the GUI, should there be one active (and if the current running speed of the program is slow enough for the GUI to keep up). When defining a new BasicInstruction in the InstructionSet class an example, the description, an instruction format, an operand mask and the simulation code are needed. The last argument to the BasicInstruction constructor, the simulation code, is what is used when the instruction is simulated.
4.2. The Mars/Mips instruction set

The guiding reference used when Mars’ instruction set was developed was *Computer Organization and Design, Fifth Edition* [14], and to see exactly which instructions are available see Appendix A. Why the instruction set is of interest is because of the goal of having Mars execute machine-generated Mips assembly.

In addition to the instructions mentioned in Appendix A there are also approximately 370 pseudo-instructions and 39 system calls used for file I/O, MIDI output, random number generation and more [4]. A pseudo instruction is similar to a macro in C. They are defined in the PseudoOps.txt file in the main folder of Mars and are expanded to their full implementation once seen in the source code.

4.3. The magic of Mars’ system calls

In regular Mips architectures the syscall instruction triggers a system call exception where the kernel takes control and handles the system call according to the value in register $v0. One downside of Mars is that the system calls are somewhat of a magic box, taking inputs, returning outputs, and causing side effects. The execution of the system call can not be followed, meaning that how the registers are updated etc. is not shown to the user. Although this is quite similar to regular Mips, where the kernel is taking over the execution from user mode, it is not very informative on what exactly is going on when a system call is made.

Although not possible to implement own system calls in Mars without altering the source code, it is possible to implement a substitute with use of the kernel data and text segments (.kdata and .ktext). By defining handlers in these segments and causing exceptions with the trap equal immediate (teqi) instruction, it is possible to simulate a more Mips-like system call compared to what Mars offers. Since this method only moves execution to another label in the source, and does not use the built-in system call, it is possible to follow the whole execution.
5. Extending Mars to execute assembly code from third party

One way to increase understanding of assembly programming would be to compile C code into Mips assembly, load it into Mars and execute it. The main challenge with this is finding a suitable compiler that can be used for the compilation. This is not an easy task as regular compilers such as GCC or Clang usually is built with the host machine as target architecture. For these compilers to compile C into Mips assembly would require to build the compiler as a cross-compiler with Mips assembly as the target architecture, something that is out of the scope for this work.

There are some other alternatives though, which do not need to be compiled just for Mips 32 assembly. The web-compiler used for this work, Compiler Explorer, works just fine as Mars can execute most of the assembly generated. Some changes needs to be done to the generated assembly, these will be listed under 'Instructions not recognized by Mars'.

Another difference with the compiled code from Compiler Explorer is that all the registers are called by numbers instead of names (for example $0 instead of $zero) and that the labels generated (besides the subroutines) have non-descriptive names. This is not optimal for new assembly programmers who are just starting out, but the register names can be seen in the Mars register overview together with the register numbers and the generated label names can be followed even though they are not descriptive.

5.1. Instructions not recognized by Mars

Some instructions generated by the web compiler will not be recognized by Mars. The instructions found not to work, and how to get them to work, will be listed here.

j - Mars does not use the jump instruction, it uses jump register jr which works the same way. Since Mars has the same functionality but with a different name, it is easy to fix this problem with pseudo instructions. To add a pseudo-instruction into Mars, add the new instruction name and format, followed by a tab-character and the instructions which are supposed to be run when the pseudo instruction is called into the PseudoOps.txt file. In the j instructions case, it will look like the following:

\[
\text{j }$t1<tab>jr\text{ RG1<tab>#Jump : Jump unconditionally to address in}\text{ $t1 (same as jr }$t1)\]

The comment followed by the hashtag (#) will be the tooltip that is displayed in the editor.

lui with registers - The compiler will generate a lui, load upper immediate, followed
by an \texttt{addiu}, add immediate unsigned with overflow, if the program is using strings. There is a pseudo instruction that can be used instead, \texttt{la}. Simply exchange the two instructions with a \texttt{la} and use the target label as address.

\texttt{slt} and \texttt{slti} - The compiler uses the \texttt{slt} - set less than - instruction both with immediate values and with registers. In Mars these two are separate, where \texttt{slti} is used for immediate values and \texttt{slt} is used for branches.

Other changes needed is to insert a \texttt{.globl main} at the start of the program to let Mars know to start execution at the main label. If this is not done, Mars will start execution at the top which will generate errors. If any strings are used in the program all of these need to be put under a data segment, and the remaining assembly code in a text segment. The last \texttt{j $31} in the main label also needs to be changed. It tries to return to the caller, but in Mars there is no caller since the function is executed at startup. The proper way to exit a program in Mars is by loading the exit syscall code 10 into register $v0 followed by a syscall.

\section*{5.2. Deciding on a calling convention}

As there is no one calling convention to rule them all, one must decide for oneself which convention to use when programming in assembly. If, for instance, the programmer chooses to save arguments for subroutines in registers instead of using the stack, then any other assembly code (like that generated from third party) must be changed so that it sticks to the same convention.

The calling convention of Compiler Explorer uses the stack to send arguments to subroutines. When defining an empty function in the C code, and later calling that function with arguments, all the arguments will be properly loaded from the stack into registers and can later be used as needed. If a programmer wants to write one function in assembly and the rest of the code in C, then this approach can be used where an empty function declaration is used in the C code. Once the C code is compiled and copied into Mars, the implementation of the assembly function can be added.
6. Extending Mars to execute C code

Allowing students to intermix code written in C together with code in Mips assembly could help bridge the gap between low level abstraction programming and languages on a higher level. The possibilities given with this extension is that from the start, students can write almost all implementation in C with only the driver code in assembly. Later, this C code can be translated into Mips assembly, keeping the programs behaviour throughout the implementation process. Depending on the functionality of the C code, the students will then learn the corresponding Mips assembly instructions needed to keep the same behaviour.

Extending Mars to run code written in C can be done in a few different ways. The first and best option would be to build a cross-compiler to compile the source into Mips assembly and then run it in Mars. If the code is compiled into Mips assembly and ran in Mars, the execution of the code could be followed. This is quite similar to the approach described in Section 5 since both approaches needs to compile C code into assembly first. Another way is to extend Mars so that it can parse and assemble C as well as assembly code, much like building a compiler inside Mars. The third and also chosen route is to enable Mars (or Java, since Mars is written in Java) to load a native library with C code.

The first idea was to create a C-file, compile it, and then run the compiled binary as a separate process (not in Mars). This is quite easy to accomplish with help of-Javas ProcessBuilder, which can be used to run the C-compiler GCC. After that, a new process can be started with the compiled binary together with chosen inputs to the C-program. This approach has one big caveat: the C-code can not return any value. If the C function can not return any value it loses its value since the function cannot be called as a sub-routine, and therefore this approach can not be followed.

6.1. An introduction to JNI

Another method of running C code which could return values is needed. In comes JNI, or the Java Native Interface [15]. JNI is an interface developed and maintained by Oracle which aims to give Java applications tools for executing native code, mainly written in C or C++. It can be used by declaring a function as native in Java and later load a native library which contains the implementation.

From JDK 8, the compiler directive you need to know about is `javac -h .` (take note of the '.', which lets the compiler know to search for source code in the current folder). The `-h` flag creates a C/C++ header file that corresponds to any methods
declared as `native` in the Java source code. Afterwards, a C file is created to provide the implementation of the generated declaration. Once both the `.c` and the header file are ready they can be compiled into a shared library (.so file for Linux and .dylib file for Mac OS), which is done with help of a C compiler of choice. For the GCC compiler on Mac OS, a few arguments must be declared:

- `-I $JAVA_HOME/include` and `-I $JAVA_HOME/include/darwin` - Tell GCC to search these folder for includes, such as "jni.h" and others that "jni.h" needs.
- `-shared` - Since this is a shared library and not an executable.
- `-o libXXX.dylib` - The name of the output file should be libXXX.dylib, where XXX is the name of the library. The naming is determined by the file structure of the project.
- The C source code file which you want to compile.

When starting up the JVM it might be needed to specify where the shared library file is located and in that case the argument `-Djava.library.path=. // (the dot here to specify the current directory) can be used when starting up the JVM. The library can be loaded into the JVM with help of the `System.loadLibrary(filename)` method, and once that is done the native C/C++ functions can be called as part of a Java program.

### 6.2. Limitations

Automating the process of creating a shared library and use it is a challenge. This is because JNI is supposed to be used by first compiling the native functions with the `javac` compiler and later give the C implementation. The problems when automating is knowing the return types as well as the arguments for the native functions beforehand. The same goes for the number of native functions. All native functions needs to be present in the Java source, and can not be entered automatically when the program is run like the native implementations can.

The extension of Mars is therefore limited to implementing one function, with integer arguments. This could then later be further developed to be fully functioning with support for multiple functions and variable arguments.

The use of JNI and how dynamic linking of shared libraries work also imposes some limitations on the project. When a library is loaded into a running JVM, it is not possible to unload it with a command the same way it is loaded with `System.load()` or `System.loadLibrary()`. For this project, where libraries are created, compiled and loaded while the program is running, this means that the first library, or the first time
a C function is assembled and loaded, is also the last for that program run. So if the programmer changes the C part of the code and re-assembles the program, the new changes will not take effect since the JVM will not load the new library. Calling the same function with different arguments will work though since it still is the same library that is used only with different data.

6.3. Alterations made to the Mars source code

To extend Mars with support for native C code through JNI, the approach used was to have all the C code at the beginning of the file and look for identifiers that starts and ends the C code section. The comment-like identifiers \#start_c and \#end_c were chosen.

6.3.1. The Tokenizer class

As described in Section 4 the Tokenizer class does pre-processing on the files before anything is modified. It is therefore easy to add yet another pre-processing method, just before the existing pre-processing is about to begin. The Tokenizer now checks whether the first line of the file is the identifier \#start_c and if it is, then it continues to parse the coming lines until an \#end_c is found. The original file without the C code is then returned, resulting in that the C code is separated from the source and can be used as needed.

6.3.2. The MIPS program class

Since the MIPS program class holds information about the current file being assembled, it fitted nicely to also store information about any C code in it. New attributes, cCode and cCompiler were added so that the code later can be compiled and run. If the Tokenizer finds any C-code in the current assembly program, both of these two new attributes are initialized to the corresponding data.

6.4. Additions made to the Mars source code

As discussed in Section 3, the addition to Mars was built and tested in isolation before integration. This new part, CompileC, does the bulk of the work with creating source and header files, compiling and linking them, and later calls them via JNI. This can be seen in Figure 4, where the code is first extracted and written to files with help of
the new `start_c` and `end_c` identifiers, compiled with GCC, and lastly loaded into the running JVM.

![Flowchart of how the C code is extracted, compiled, and loaded.](image)

**Figure 4:** Flowchart of how the C code is extracted, compiled, and loaded.

### 6.4.1. The CompileC class

When the MIPS program class has received all C code from a file it creates a new `CompileC` object which is stored as an attribute. When the `CompileC` object is created, all C-code from the assembly program is entered together with any other code which is needed for the program to work correctly such as extraction of arguments. Since it is not known beforehand how many arguments each C function has, the native Java (C) function is defined to take an array of integer arguments. When the C function later is parsed, all the arguments get extracted from the argument array and named according to the C code from the assembly file.

The `CompileC` class has three methods which are used to call a C function from assembly: `createCFileAndHeaderFile`, `compileCCode` and `executeCCode`. The first method uses Java’s `FileWriter` class to create the C and header file, filled with the source code, the second uses Java’s `ProcessBuilder` to run GCC and compile the generated source code into a shared library and the third simply executes the native function with a given array of integer arguments.

### 6.4.2. Implementing a new system call

Once the C functions source code is created, compiled and loaded into the running JVM the next objective is to figure out a way of calling this function. As described in Section 4, the Mars system calls is quite magical with all of the implementations in Java. All of the 39 system calls available in Mars are implemented in the `mars/mips/instructions/syscalls` folder and they all implement the `AbstractSyscall` interface, which except the constructor has one method, `simulate`. The `simulate` method is run when the system call is used and is implemented in Java, not using any Mips assembly instructions.
In order to follow the convention of system calls, it was chosen to also implement function calls to C with a system call. The first vacant system call number, 18, was chosen and its simulate method simply calls the MIPS program currently running's executeCCode method. Still following the Mips convention, the system call expects all arguments to the function to be in registers $a0$–$a3$ and the return value is placed in register $v0$. 
7. Result

The results of this project is, just like its implementation, divided into two parts: Extending Mars to run assembly from third party and extending Mars to execute C code.

7.1. Extending Mars to execute assembly code from third party

The code generated by the web compiler Compiler Explorer as seen in Figure 11a and 11b can be executed in Mars, with a few modifications. Firstly, the \texttt{j \$31} at the end of the main label needs to be changed to \texttt{li \$v0, 10} followed by a system call. As discussed earlier this is because register 31 will be empty since the main routine does not have a caller. The proper way to exit a program in Mars is by putting the correct code, 10, in register \texttt{\$v0} followed by a system call.

Another problem with the compiled code is that it will start executing from the top, at the \texttt{.add} label. When execution comes to the jump back to the calling routine, it will exit with errors because there is no routine to return to. To solve this problem, either switch the order of the labels so that the main label comes first, or add a \texttt{.globl main} to the top of the file to let Mars know where to start execution.

Using functions that are imported from other header files is generally not a good idea, since when compiling a C program with imported header files the imported functions will be fetched and included in the binary. For example, if the header file \texttt{<stdio.h>} is imported and a call to \texttt{printf} is made, the corresponding Mips assembly will be a jump and link (\texttt{jal}) to \texttt{printf}. For this to work in Mars an implementation of the \texttt{printf} function must be added, either in a separate file or in the same file where the call is made. It is of course possible, although time-consuming, to find the implementations of all the library functions that are called. It is however advised to stick to standard library functions or to find corresponding functionality in Mars. For the \texttt{printf} example there is a quick and easy solution: use the print integer, float, double, or string system call instead.

There are three main different approaches to working with the web compiler - writing all the code in C and running the corresponding assembly in Mars, writing most of the functionality in assembly and only write one function in C, and writing most of the functionality in C with the addition of one function in assembly. How to run a whole C program in Mars has already been covered, so now will the other two approaches be described.
7 RESULT

7.1.1. Calling assembly function from C

If you are going to write a single function in assembly and having the rest of the implementation in C, you will have to use the same calling convention in your assembly function as is used for the remaining of the program. One way of working with this is described in Section 5, namely to make use of empty function declarations. As seen in Figure 12a and 12b which is the same example as the first two, the empty function declaration of add still gives some useful assembly code. If the programmer wants to write its own implementation of the add function on assembly level, it is possible to do so after the first nop in the add subroutine. The difficulties of the calling convention are thereby overcome since the variables used (int a, int b) is loaded into registers $4$ and $5$ respectively. The return value can then be placed in register 2, or $v0$, since both $v0$ and $v1$ are supposed to be used for subroutine return values.

7.1.2. Calling C function from assembly

When writing a single function in C there are two options: using the same calling convention as the web-compiler does, or use your own and change the resulting assembly code from the web compiler. As previously mentioned, the web compiler uses the stack to send and receive arguments to and from subroutines. This means that the remaining assembly code either needs to do the same, or the compiled assembly needs to be changed. Depending on the complexity of the C function, it might be easier to change so that the corresponding assembly uses registers instead of the stack. For instance, if the function does not need all registers, and thereby do not need to save to the stack before starting execution, the stack manipulating of the generated assembly can be removed and the input can simply be put in some of the available registers ($a0 - a3$). The reason for only doing this when the code is uncomplicated is that more intricate code usually has a lot of frame and stack pointer manipulation, making it hard to exchange them all without changing the behavior of the program.

7.2. Extending Mars to execute C code

Extending Mars to execute code written in C has been fully implemented according to the limitations set. It is possible to write one function in C, with a variable number of integer parameters, and execute it via the newly implemented system call. Figure 5 shows a trivial example use of the new functionality and a more intricate example can be found in appendix C, Figure 14.

The function written in C must be enclosed with the new directives #start_c and #end_c, with #start_c on the first line of the file. The functions parameters should be
placed in registers $a0 - $a3 and the return value will be placed in register $v0. The function can then be called by loading the new system call code 18 into register $v0 followed by an issued system call. The C source code with the function implementation is created, compiled, and loaded upon assembling of the program and the function call via JNI is done upon execution of the system call.

```assembly
1 #start_c
2 int foo(int x){
3     return x+42;
4 }
5 #end_c
6 main:
7     li $a0, 10
8     li $v0, 18
9     syscall
10    li $v0, 10
11    syscall
```

**Figure 5:** Example use of C in Mars.

The example program in Figure 5 starts by opening a block of C-code with the directive #start_c on line 1. The function declared takes an integer input and returns that integer plus 42, and the block is then closed with the directive #end_c. In the main label of the assembly code, an argument to the C function is loaded into register $a0 on line 7, followed by loading the new system call code 18 into register $v0 on line 8. When the system call then is made on line 9, the C function is executed with the argument in register $a0, namely 10. The return value, 10+42, is available in register $v0. After that, the program is finished and is exited by loading the exit system call code 10 into register $v0 followed by a system call.

It is recommended to write functioning C code separately and insert it into Mars since the editor is developed to support Mips assembly code, not C code. The syntax highlighting does not work properly for C code and it is generally not a very pleasant experience to write C code in Mars.

### 7.2.1. Shortcomings

Due to the limitations of the JNI interface discussed in Section 6, it is not possible to first assemble a program with C code, make changes to the C code, and then assemble the program again. When this is done, only the first version of the C code will be loaded as a library function. This is because JNI can not have multiple loaded functions with the same library and function name and that it is impossible to manually unload a library from the JVM.
8. Discussion

The results of both approaches followed in this thesis yielded fruit. It is possible to both run compiled assembly from third party in Mars, as well as execute intermixed assembly and C code. Following are some use cases where the different approaches suits best for educational use, together with a comparison of the two and some concrete examples that can be used when teaching.

8.1. Extending Mars to execute C code

Executing C code in Mars would be most beneficial in the very early stages of learning assembly programming. Assuming that the students already knows C, would it be possible for them to instead of writing a function in assembly write the function in C instead. Then all that needs to be done is call the C function from the assembly part of the code, something that could serve as a bridge into learning assembly. For calling C functions from assembly and printing the output to the console, it is only needed to know about three different system calls and how to use them. This will serve as an introduction to system calls as well as the assembly syntax and Mips registers. Another way executing C code in Mars can be used is by starting off with a program that has a large part of its implementation in the C code, and then gradually translating from C to assembly, finishing with pure assembly.

One rather big downside to executing C code in Mars is that the execution part is a black box, with no insight into what happens in the registers. Arguments are given and output is received but students are not able to see the actual execution, and it would be far better to visually display, even when executing C code, what happens in the registers and various segments.

8.2. Extending Mars to execute assembly from third party

Compiling C code into assembly and executing it in Mars can be used either as a first step towards assembly programming, or when the students have started to gain some knowledge. It can also be used just for learning the calling convention used by the web compiler, where all the arguments to functions are saved on the stack before jumping to a subroutine. It can however also be used later in the teaching since it will depend on the level of intricacy that the C program compiled and ran has or if compiler optimizations, for instance, is being used.
8.3. Comparison of the two approaches

Both approaches could serve a purpose in aiding students learning of assembly programming. One difference with executing C code in Mars compared to executing assembly from third party is that executing C code in Mars most likely only has educational gain during the very early stages of students learning. First compiling C code to assembly and then executing it in Mars however, could help students far longer. Only using the web compiler used in this project could very well be of significance, since the students can clearly see how the C code is compiled and what program statement gets translated to what.

Considering that executing C code is like a black box, it would be better to first translate the C code into Mips assembly, and then execute it as normal in Mars. Since that is not possible with the current implementation, it is better to make use of the web compiler to translate the C code into assembly and then execute it. Even though the students will need an additional tool besides Mars, they gain a very good oversight of how the program is compiled and also run when executed in Mars.

8.4. Concrete examples

To show how the approaches explored in this thesis can be used in teaching of assembly programming is following three examples. They will start rather easily and then increase in difficulty, and one or both approaches will be used.

8.4.1. Example 1: Study program with various instructions

When starting to learn assembly programming it could be enough to just examine an assembly program. Here students are tasked with entering a rather simple C program such as Figure 6a into the web compiler and then copying the resulting assembly, Figure 6b, to Mars and running it. The program should make use of different standard functionalities such as addition, subtraction, multiplication and can also be made more difficult with additions of if/else clauses, calls to subroutines, and loops. The goal will be to familiarise the students with many usual instructions for the Mips architecture as well as Mars.
8.4.2. Example 2: Understanding for-loops

The students can start by either writing a rather simple C program containing a for-loop or receiving it from the teacher, enter it into Mars, and calling it with appropriate values and system calls as seen in Figure 7. They can then continue to move out the loop body to the assembly code, and in every iteration of the loop call the C function which now only returns \(x+5\). Once that works as intended they can remove the C part of the code completely and integrate the last part, incrementing the variable \(x\) by 5, into the for-loop.
8.4.3. Example 3: Understanding stack manipulation with sub routines

Let the students go to the web compiler, choosing C to MIPS gcc 5.4 compilation. There they should enter a function with two or more arguments, like in Figure 8, and copy the resulting assembly code into Mars. The task will then be to write assembly which calls the generated subroutine with proper arguments, using the same calling convention that the compiled assembly does.
```c
int add(int a, int b){
    return a+b;
}
```

(a) Add function in C.

```assembly
1 add:
2    addiu $sp,$sp,-8
3    sw $fp,4($sp)
4    move $fp,$sp
5    sw $4,8($fp)
6    sw $5,12($fp)
7    lw $3,8($fp)
8    lw $2,12($fp)
9    nop
10   addu $2,$3,$2
11   move $sp,$fp
12   lw $fp,4($sp)
13   addiu $sp,$sp,8
14   j $31
15   nop
```

(b) Corresponding function in assembly.

**Figure 8:** Learning stack manipulation with sub routines.
9. Conclusion

This thesis investigates tools for teaching students assembly programming. One already used tool, Mars, has been extended with support for executing code written in C, giving students the ability to go from something familiar to something new. A new tool has also been introduced, the web-compiler Compiler Explorer. This new tool gives students a very detailed translation of C to Mips assembly with color coding and the ability for extra compiler directives. That Mars is able to execute C code has some area of use, but giving students the ability to first compile the C code to assembly and then execute it in Mars seems to enhance students learning the most.

The vision of having Mars as an all-in-one tool for learning assembly does seem feasible after this thesis. If a cross-compiler such as GCC or Clang were to be built into Mars and the new functionality of having two tabs open at the same time where added, the goal would almost be achieved. All that would be left is adding a new button for generating Mips assembly from C source and color-coding. If the compiler is run with verbose arguments the color-coding might not even be needed, since it will generate comments in the compiled assembly.
10. Future work

Considering that this project is more of a pre-study trying to find new techniques for learning assembly programming, there are still quite a few additions that could further enhance students learning. These will be discussed here, together with what needs to be done in order to achieve vision of Mars discussed earlier.

10.1. Integrating a C to Mips 32 compiler in Mars

In order to move from a web-based tool, such as Compiler Explorer which is used for this project, to an integrated one, it would be needed to build a cross compiler and integrate it into Mars. That way everything could be done inside Mars and the need for additional tools would be removed. Extra functionality such as color coding and extra tabs could also be implemented, making Mars a complete all-in-one tool for learning assembly programming. The web-compiler used in this project does implement just that, but having every utility in one place would make it easier to use. Mars does already support tabs while editing source code, but it is only possible to have one tab open at the same time. It would be better if it was possible to have two tabs, one with C code and the other with corresponding Mips assembly code.

10.2. Full implementation of C code functionality in Mars

This work only implemented support for one C function with variable integer parameters. A future extension could be if the implementation could support arguments of different types and also multiple functions.

Also connected to running C code in Mars are the JNI limitations. The implementation can be further developed with better function parsing (support for multiple functions) and the ability to unload libraries from the JVM. Although not possible with any framework functionality, it might be possible by making sure that the class which loaded the library is garbage collected before a new library is loaded.

10.3. Changing register names of compiled assembly

It is a lot to take in when learning assembly programming, especially in the beginning. Using a tool that then uses a different naming convention of the registers, even if it is possible in Mars to see the more widely used ones next to them, might be a little confusing. It would therefore be better if the compiled code would use the same naming
convention that Mars originally does, with letters and numbers ($v0, v1$) instead of just numbers, ($2, 3$). It would be possible, and not very extensive, to perhaps add a new button in Mars that when pressed parses the assembly code and exchanges numbers to letters and numbers.

10.4. Automating manual work

If a parser is implemented for changing the names of registers, that parser could also be extended with functionality for doing some of the manual work that is needed in order to run the cross-compiled assembly in Mars. Right now the programmer must change the last \texttt{j $31} in the end of the main label as well as the other instructions discussed in section 5, something that could be automated as a middle-step between the prototype that is built in this work and the final vision of Mars.

10.5. Survey on students learning

To really evaluate the result of this project it would be best to do a survey on students who are learning assembly programming. One way could be to let one group of students use the current learning tools, and let one group also use the tools from this thesis. Then it would be possible to evaluate the performance of students based on their grades and understanding of the course.

10.6. Implementing the vision of Mars

In order to fulfill the vision of Mars described in various sections of this thesis, the first action should be to integrate a cross-compiler into Mars. Once that is done, support for having two separate tabs, one with the C source and once with the generated assembly is needed. When that works as intended and Mars only executes the Mips assembly, a new button needs to be added to compile the C source into assembly and exchange the register names at the same time. As previously mentioned, color coding might not be necessary if the \texttt{-fverbose-asm} flag is used when compiling.

Once all those pieces are in place the Mars simulator is, according to this thesis goals, optimal.
References


A. MIPS Instruction Set

This appendix holds the guiding reference used when developing Mars.

<table>
<thead>
<tr>
<th>MIPS Core Instructions</th>
<th>Name</th>
<th>Format</th>
<th>MIPS Arithmetic Core</th>
<th>Name</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add</td>
<td>R</td>
<td>multiply</td>
<td>mul</td>
<td>R</td>
</tr>
<tr>
<td>add immediate</td>
<td>addi</td>
<td>I</td>
<td>multiply unsigned</td>
<td>multiu</td>
<td>R</td>
</tr>
<tr>
<td>add unsigned</td>
<td>addu</td>
<td>R</td>
<td>divide</td>
<td>divu</td>
<td>R</td>
</tr>
<tr>
<td>add immediate unsigned</td>
<td>addiu</td>
<td>I</td>
<td>divide unsigned</td>
<td>divu</td>
<td>R</td>
</tr>
<tr>
<td>subtract</td>
<td>sub</td>
<td>R</td>
<td>move from hi</td>
<td>mfhi</td>
<td>R</td>
</tr>
<tr>
<td>subtract unsigned</td>
<td>subu</td>
<td>R</td>
<td>move from lo</td>
<td>mflo</td>
<td>R</td>
</tr>
<tr>
<td>AND</td>
<td>and</td>
<td>R</td>
<td>move from system cont. (EPC)</td>
<td>mfc0</td>
<td>R</td>
</tr>
<tr>
<td>AND immediate</td>
<td>andi</td>
<td>I</td>
<td>floating point add single</td>
<td>add.s</td>
<td>R</td>
</tr>
<tr>
<td>OR</td>
<td>or</td>
<td>R</td>
<td>floating point add single</td>
<td>add.d</td>
<td>R</td>
</tr>
<tr>
<td>OR immediate</td>
<td>ori</td>
<td>I</td>
<td>floating point add single</td>
<td>sub.s</td>
<td>R</td>
</tr>
<tr>
<td>NOT</td>
<td>nore</td>
<td>R</td>
<td>floating point add single</td>
<td>sub.d</td>
<td>R</td>
</tr>
<tr>
<td>shift left logical</td>
<td>slt</td>
<td>R</td>
<td>floating point multiply single</td>
<td>mul.s</td>
<td>R</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srli</td>
<td>R</td>
<td>floating point multiply single</td>
<td>mul.d</td>
<td>R</td>
</tr>
<tr>
<td>load upper immediate</td>
<td>lui</td>
<td>I</td>
<td>floating point divide single</td>
<td>div.s</td>
<td>R</td>
</tr>
<tr>
<td>load word</td>
<td>lw</td>
<td>I</td>
<td>floating point divide double</td>
<td>div.d</td>
<td>R</td>
</tr>
<tr>
<td>store word</td>
<td>sw</td>
<td>I</td>
<td>load word to floating point single</td>
<td>swc</td>
<td>I</td>
</tr>
<tr>
<td>load halfword unsigned</td>
<td>lhu</td>
<td>I</td>
<td>load word to floating point single</td>
<td>swcl</td>
<td>I</td>
</tr>
<tr>
<td>store halfword</td>
<td>sh</td>
<td>I</td>
<td>load word to floating point double</td>
<td>swcl</td>
<td>I</td>
</tr>
<tr>
<td>load byte unsigned</td>
<td>lbu</td>
<td>I</td>
<td>load word to floating point double</td>
<td>slcl</td>
<td>I</td>
</tr>
<tr>
<td>store byte</td>
<td>sb</td>
<td>I</td>
<td>branch on floating point true</td>
<td>bct</td>
<td>I</td>
</tr>
<tr>
<td>load linked (atomic update)</td>
<td>li</td>
<td>I</td>
<td>branch on floating point false</td>
<td>bclr</td>
<td>I</td>
</tr>
<tr>
<td>store cond. (atomic update)</td>
<td>sc</td>
<td>I</td>
<td>floating point compare single</td>
<td>cxs</td>
<td>R</td>
</tr>
<tr>
<td>branch on equal</td>
<td>beq</td>
<td>I</td>
<td>(x = eq, ne, ltl, le, gt, ge)</td>
<td>cxd</td>
<td>R</td>
</tr>
<tr>
<td>branch on not equal</td>
<td>bne</td>
<td>I</td>
<td>floating point compare double</td>
<td>cxd</td>
<td>R</td>
</tr>
<tr>
<td>jump</td>
<td>j</td>
<td>J</td>
<td>(x = eq, ne, ltl, le, gt, ge)</td>
<td>cxd</td>
<td>R</td>
</tr>
<tr>
<td>jump and link</td>
<td>jal</td>
<td>J</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jump register</td>
<td>jr</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set less than</td>
<td>slt</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set less than immediate</td>
<td>slti</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set less than unsigned</td>
<td>situ</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set less than immediate unsigned</td>
<td>sitlu</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 9:** MIPS instruction set from Computer Organization and Design, Fifth Edition

---

### 3.10 Concluding Remarks

<table>
<thead>
<tr>
<th>Name</th>
<th>Format</th>
<th>Pseudo MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>I</td>
<td>load word</td>
</tr>
<tr>
<td>lwl</td>
<td>I</td>
<td>load address</td>
</tr>
<tr>
<td>lwr</td>
<td>I</td>
<td>load double</td>
</tr>
<tr>
<td>sw</td>
<td>I</td>
<td>store word</td>
</tr>
<tr>
<td>swl</td>
<td>I</td>
<td>store double</td>
</tr>
<tr>
<td>swr</td>
<td>I</td>
<td>store word</td>
</tr>
<tr>
<td>lui</td>
<td>I</td>
<td>load immediate</td>
</tr>
<tr>
<td>sc</td>
<td>I</td>
<td>store content</td>
</tr>
<tr>
<td>move</td>
<td>R</td>
<td>move word</td>
</tr>
<tr>
<td>movev</td>
<td>R</td>
<td>move word</td>
</tr>
<tr>
<td>beqz</td>
<td>R</td>
<td>branch on equal zero</td>
</tr>
<tr>
<td>bnez</td>
<td>R</td>
<td>branch on unequal zero</td>
</tr>
<tr>
<td>bgez</td>
<td>I</td>
<td>branch on greater than or equal</td>
</tr>
<tr>
<td>bgtz</td>
<td>I</td>
<td>branch on greater than</td>
</tr>
<tr>
<td>blez</td>
<td>I</td>
<td>branch on less than or equal</td>
</tr>
<tr>
<td>bltz</td>
<td>I</td>
<td>branch on less than</td>
</tr>
<tr>
<td>bx</td>
<td>I</td>
<td>branch on zero</td>
</tr>
<tr>
<td>brk</td>
<td>R</td>
<td>branch on key</td>
</tr>
<tr>
<td>trap</td>
<td>R</td>
<td>trap immediate</td>
</tr>
</tbody>
</table>

*Note: The format for each instruction is shown as 'I' for immediate, 'R' for register, 'I' for instruction, and 'I' for immediate.*

Figure 3.27 Remaining MIPS-32 and Pseudo MIPS instruction sets. f means single (s) or double (d) precision floating point instructions, and s means signed and unsigned (u) versions. MIPS-32 also has FP instructions for multiply and add/sub (maddd, fmad, fsub), ceiling (ceilf), truncate (truncf), round (rundf), and reciprocal (recipf). The underscore represents the letter to include to represent that datatype.

Figure 10: Remaining Mips instruction set and pseudo instructions from Computer Organization and Design, Fifth Edition²

B. Examples of C and Assembly code

Here are some examples of C code with its corresponding assembly code generated by the web-compiler used in this thesis.

B.1. Compilation from C to Mips assembly

```
int add(int a, int b)
{
    return a + b;
}

int main(void){
    int x = 5;
    int y = 10;
    int c = add(x,y);
    return 0;
}
```

```
add:  addiu $sp,$sp,-8
       sw  $fp,4($sp)
       move $fp,$sp
       sw  $4,8($fp)
       sw  $5,12($fp)
       lw  $3,8($fp)
       lw  $2,12($fp)
       nop
       addu $2,$3,$2
       move $sp,$fp
       lw  $fp,4($sp)
       addiu $sp,$sp,8
       jal add
       nop
       addiu $sp,$sp,-48
       sw  $31,44($sp)
       sw  $fp,40($sp)
       move $fp,$sp
       li  $2,5
       sw  $2,24($fp)
       li  $2,10
       sw  $2,28($fp)
       lw  $5,28($fp)
       lw  $4,24($fp)
       jal add
       nop
       sw  $2,32($fp)
       move $2,0
       move $sp,$fp
       lw  $31,44($sp)
       lw  $fp,40($sp)
       addiu $sp,$sp,48
       j  $31
       nop
```

(a) Simple C program with function call.  
(b) Generated assembly.

**Figure 11**: Translation of C to assembly.
B.2. Use of empty function declarations in C

```c
int add(int a, int b){
    }
int main(void){
    int x = 5;
    int y = 10;
    int c = add(x, y);
    return 0;
}
```

```assembly
add:  addiu $sp, $sp, -8
       sw $fp, 4($sp)
move $fp, $sp
sw $4, 8($fp)
sw $5, 12($fp)
nop
move $sp, $fp
sw $4, 8($fp)
addiu $sp, $sp, 8
j $31
nop
main:
addiu $sp, $sp, -48
sw $31, 44($sp)
sw $fp, 40($sp)
mov $fp, $sp
li $2, 5
sw $2, 24($fp)
li $2, 10
sw $2, 28($fp)
lw $5, 28($fp)
lw $4, 24($fp)
jal add
nop
sw $2, 32($fp)
mov $2, $0
mov $sp, $fp
sw $31, 44($sp)
lw $fp, 40($sp)
addiu $sp, $sp, 48
j $31
nop
```

(a) Empty C function.

(b) Empty function in assembly.

**Figure 12:** Use of empty functions in C.
C. Intermixed C and assembly code

Here are some examples on how the Mars can be used to execute intermixed C and assembly code.

C.1. Trivial example

```c
#include <stdio.h>

int foo(int x)
{
    return x+42;
}
```

```assembly
main:
li $a0, 10
li $v0, 18
syscall
li $v0, 10
syscall
```

Figure 13: Trivial C function executed from assembly.
C.2. N:th Fibonacci number

```c
#start_c
int fibonacci(int n){
    int i;
    int f1 = 0;
    int f2 = 1;
    int fi;
    if(n == 0)
        return 0;
    if(n == 1)
        return 1;
    for(i = 2 ; i <= n ; i++ )
    {
        fi = f1 + f2;
        f1 = f2;
        f2 = fi;
    }
    return fi;
}
#end_c

main:
    #argument to fibonacci function
    li $a0, 30
    #syscall code for c function call
    li $v0, 18
    syscall
    #move result of C function to a0
    move $a0, $v0
    #syscall code for print integer
    li $v0, 1
    syscall
    #syscall code for exit
    li $v0, 10
    syscall
```

**Figure 14:** Assembly program to print n:th fibonacci number.