Hardware Agnostic Programming of Embedded Systems

Peter Otrebus-Larsson
2015

Master of Science in Engineering Technology
Computer Science and Engineering

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering
Abstract

This thesis presents a solution to the problem of programming a discrete embedded system whose hardware architecture is unknown. A programming interface standard is proposed which when implemented on the host system presents it as a remotely programmable virtual machine with real-time concurrency and scheduling support based on the Timber model of reactive objects.

Using the TinyTimber API, the standard was realized on an AVR-based host platform. On a client computer, a compiler from a high-level language to the instruction set of the virtual machine was implemented, which was used to demonstrate the suitability of the proposed standard in terms of usability and expressive power. The execution performance of the virtual machine was measured to compare favorably with other interpreted virtual machines.
Preface

This work was carried out for Neava Consulting AB in Luleå between May and October 2014. I wish to thank my external supervisor Staffan Johansson for his time and ideas and for providing me with the necessary equipment to carry out this work. My gratitude also extends to my internal supervisor Jens Eliasson for his feedback and encouragement.

At the time of writing, the source code constituting the implementation of this work can be found at https://github.com/Otrebus. This report was typeset in \LaTeX{} and the figures were made with Ti\textit{k}Z.
# Contents

1 Introduction .......................................................... 1
   1.1 Background and problem statement .............................. 1
       1.1.1 Reprogrammability .................................... 1
       1.1.2 Hardware abstraction ................................ 1
   1.2 Purpose and solution overview .................................... 2
   1.3 Delimitations ....................................................... 2
   1.4 Related work ....................................................... 3

2 Theory ................................................................. 4
   2.1 The TinyTimber model of reactive objects ....................... 4
       2.1.1 Types of method invocations ............................ 4
       2.1.2 Scheduling ............................................... 5
       2.1.3 Some implications ....................................... 6
   2.2 Virtual machines .................................................. 6
   2.3 Compiler construction .............................................. 7
       2.3.1 Lexing ................................................... 8
       2.3.2 Parsing .................................................. 8
       2.3.3 Code generation ......................................... 9
       2.3.4 Intermediate code and other steps ...................... 10

3 Design ................................................................. 11
   3.1 Virtual machine design .......................................... 11
       3.1.1 Execution environment overview ......................... 11
       3.1.2 Instruction set overview ................................ 12
       3.1.3 The anatomy of a program ............................... 13
       3.1.4 Loading a program — the binary format ............... 14
       3.1.5 Method invocation techniques, the stack frame ....... 14
   3.2 Communication design ............................................. 17
       3.2.1 The protocol for programming the virtual machine .... 18
       3.2.2 Sending application data ............................... 19
   3.3 High level language ............................................... 19
       3.3.1 Overview ................................................. 19

4 Implementation ....................................................... 22
   4.1 Hardware equipment and software tools ......................... 22
   4.2 The virtual machine ............................................. 23
       4.2.1 Linking and loading a program ......................... 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.2</td>
<td>Instruction dispatch — the exec function</td>
<td>23</td>
</tr>
<tr>
<td>4.2.3</td>
<td>External functions</td>
<td>26</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Error handling</td>
<td>26</td>
</tr>
<tr>
<td>4.3</td>
<td>The assembler</td>
<td>26</td>
</tr>
<tr>
<td>4.4</td>
<td>The compiler</td>
<td>27</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Parsing</td>
<td>27</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Symbol table construction</td>
<td>27</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Code generation</td>
<td>28</td>
</tr>
<tr>
<td>5</td>
<td>Evaluation</td>
<td>32</td>
</tr>
<tr>
<td>5.1</td>
<td>Performance</td>
<td>32</td>
</tr>
<tr>
<td>5.2</td>
<td>Expressibility</td>
<td>34</td>
</tr>
<tr>
<td>6</td>
<td>Future work</td>
<td>35</td>
</tr>
<tr>
<td>6.1</td>
<td>Error handling</td>
<td>35</td>
</tr>
<tr>
<td>6.2</td>
<td>Querying</td>
<td>35</td>
</tr>
<tr>
<td>6.3</td>
<td>Information security</td>
<td>35</td>
</tr>
<tr>
<td>6.4</td>
<td>Extended virtual machine capabilities</td>
<td>36</td>
</tr>
<tr>
<td>6.5</td>
<td>Context switch elimination</td>
<td>36</td>
</tr>
<tr>
<td>A</td>
<td>Appendix: Language grammar</td>
<td>37</td>
</tr>
<tr>
<td>B</td>
<td>Appendix: Turing machine</td>
<td>40</td>
</tr>
<tr>
<td>C</td>
<td>Appendix: Instruction set</td>
<td>42</td>
</tr>
<tr>
<td>D</td>
<td>References</td>
<td>48</td>
</tr>
</tbody>
</table>
1 | Introduction

This chapter serves to provide motivational background for the work carried out in the context of this thesis, as well as a preliminary overview of the work itself.

1.1 Background and problem statement

Embedded systems — discrete and often task-specific computer devices that are components of larger systems — are becoming increasingly ubiquitous. More than 98% of all processors employed today reside in embedded systems [1], and all around us these components carry out critical tasks in an increasing amount of technological implements with an ever growing range of applications. Indeed, it is estimated that the worldwide market for the Internet of Things sector alone will increase to exceed $7 trillion by 2020 [2].

1.1.1 Reprogrammability

The degree of specialization of a particular embedded system, or how task-specific it is, depends on the application context. For example, in the braking system of a car, a particular microcontroller is not expected to ever change its functional behavior or its general area of application. On the other hand, in the context of a sensor network it might be advantageous for each constituent system (node) to have the ability to be reprogrammed after deployment in order to be able to adapt their functionality to unforeseen situations. Imagine a scenario where a great number of small meteorological sensors with temperature meters have been released by plane over a mountainous island. They are programmed to make temperature readings every few minutes, store this information, and then transmit a summary of this information once every day. Now, if one of the mountains actually turns out to be a volcano that is expected to erupt, it might be instructive to be able to reprogram the behavior of the sensors to instead transmit temperature readings every few seconds to track the flow of lava in real-time and visualize it. The ability to do so remotely, without having to physically access each separate node, is clearly advantageous in such a scenario.

1.1.2 Hardware abstraction

An idea to be linked closely to the notion of dynamic reprogramming of a system is the concept of hardware abstraction, where the underlying architecture of some discrete device is presented through an interface that serves to provide hardware operations independent of the technical particulars of the given system. Traditionally, this interface — the hardware abstraction layer — is often exposed to the developer in the form of a linked library together with an application programming interface that allows the developer to write code that is agnostic to the internal workings of the hardware components. Still, with this approach the code needs to be compiled to a language supported by the CPU architecture of the system before uploading
it to the system (often by overwriting the flash memory), so in a sense, the outside world still needs to know the architecture of the system to be able to write code for it.

Going back to the example of the meteorological sensors and instead imagining that the described sensors have varying brands of temperature sensors and maybe even varying CPU architectures, the task of reprogramming them is more complex since different code would have to be uploaded to each different architecture. If the architecture for a given system is unknown, the task would be impossible. However, if these individual systems, or nodes, implement some sort of standardized interface for remote reprogramming, it would allow a developer to reprogram the devices without knowing about their internal particulars; only one piece of code would need to be written for all systems.

1.2 Purpose and solution overview

The purpose of the work carried out within the context of this thesis is to design and implement a programming interface standard that provides both the expressive power and hardware abstraction necessary to enable hardware-agnostic remote reprogrammability of embedded systems. Specifically, a design is presented for a programming interface whose implementation can be divided into the following parts:

- A virtual machine running on the embedded system that is programmable through a set of instructions. Programs running in the virtual machine can perform any sort of algorithmic work, and also access hardware devices exposed by the virtual machine.
- A communication interface by which to upload bytecode — programs consisting of virtual machine instructions — to the device for the virtual machine to execute.
- A compiler that translates a high level language to bytecode, for convenience.

This standard presents a unified view of different embedded systems to the outside world, and allows a developer to program these systems without knowing their specific hardware architecture.

While the problem of hardware abstraction has been addressed in earlier research yielding different types of solutions (see Section 1.4), the solution presented in this thesis attempts to provide a novel approach by partially basing the computational environment provided by the virtual machine on the Timber model of concurrent objects. This model provides the functionality for the virtual machine to carry out the time critical tasks often carried out by embedded systems, which includes task scheduling, interrupt handling and pseudo-parallel execution together with a method of concurrency control.

1.3 Delimitations

In order to conform the workload committed to this thesis to the given time constraints, some delimitations were imposed on the overall scope of the project. With a few exceptions, only the most essential mechanisms for compiling, uploading and running a program on the virtual machine were designed and implemented. Some error handling was implemented in the compiler and the transmission protocol, but no error handling was designed or implemented
in the virtual machine either in the execution stage or in the linking process. Moreover, there is currently no protocol designed for any sort of application-external communication with the virtual machine, such as querying its capabilities in terms of hardware and similar.

The areas mentioned to be out of scope of the thesis work are also addressed in Chapter 6, which mentions a number of possible future expansions of the solution presented in this thesis.

1.4 Related work

The potential virtues of using a virtual machine to provide a level of programming abstraction and dynamic reprogrammability for embedded systems has inspired a number of different approaches:

Scylla [3] is a virtual machine architecture which executes its own instruction set that is similar to the instruction set of the CPU architectures of many embedded systems, which allows it to easily translate program bytecode into actual CPU instructions "on the fly". This allows for fast execution, but instruction sets that are highly non-orthogonal or dissimilar to the Scylla instruction set would preclude this approach of direct mapping.

Another example is the Darjeeling [4] virtual machine which is inspired by the Java virtual machine and is able to execute a subset of the Java language together with automatic garbage collection. It is designed to run on 8- or 16-bit microcontrollers with a low amount of RAM, and is targeted towards nodes in sensor networks. It does not support just-in-time compilation like Scylla, but instead interprets the given byte code in software, at a performance overhead of between 30x-100x. This can be compared to the overhead reported by the Maté interpreter (35x) [5], and the SCript interpreter (10x-35x) [6].

A recent invention is the Insense [7] virtual machine which is a specialized Java virtual machine that utilizes a "split VM architecture", where class files are linked on a desktop machine and are statically compiled together with the virtual machine to be installed onto the embedded system. Dynamic reprogramming however, is not supported. Insense features a model for concurrent programming where encapsulated components communicate through message passing. The virtual machine code of the Insense virtual machine is software interpreted, like in the case of Darjeeling. The authors argue that sensor nodes, which is the targeted application area, are not expected to perform computationally intensive tasks, but rather routine I/O tasks, so the performance overhead is not a major issue.
2 Theory

In this chapter, a few relevant topics will be covered to provide some prerequisite background for the remainder of this report. Some of the topics are quite vast, so only a few aspects of each topic that are most closely associated with the scope of this thesis will be discussed. Section 2.1 provides an introduction to TinyTimber, which is a small real-time kernel that provides both the inspiration and the underpinnings of the scheduling and concurrency features of the virtual machine. Some relevant theory about virtual machines is given in Section 2.2. The general process of creating a compiler is covered in section 2.3 to give some familiarization of the parts relevant to the implementation of the compiler made for this thesis work.

2.1 The TinyTimber model of reactive objects

The full Timber programming language [8] is based on an object-functional and concurrent extension of Haskell, but for the purposes of this thesis, only the concurrency features which are implemented in the TinyTimber [9] port to C will be discussed here.

The basic conceptual building block in TinyTimber is the notion of a reactive object, similar to a C++ or Java type object with encapsulated state and callable methods, which is idle until activated by an invocation of one of its methods by either an external event or a call from a method in another object. Execution is entirely preemptive, which means that the execution of a method in one object can be temporarily halted during the execution of a new method call in another object. To provide concurrency control, objects behave like monitors in that only one method call can ever be processed by a given object at any given time. This gives the same effect as if each object had a single unique mutex that is locked at the entrance of every method, and unlocked at the exit of every method. Any other incoming call to an object is thereby delayed until the current call is finished.

2.1.1 Types of method invocations

Methods on objects can be called from other objects either synchronously or asynchronously. To illustrate, imagine that there is an object A containing a method Foo. Also assume that object A is processing some method call (not necessarily Foo), and the execution is preempted by a call to some method Bar of object B.

Synchronous calls

If the execution of Bar reaches a synchronous call to Foo, it can not immediately invoke that method since A is already locked by another call. The execution of Bar must then pause at the synchronous call until object A has finished processing its previously preempted method call. After A has finished executing, the synchronous statement in Bar can finally invoke Foo and use a possible return value of that method. See Figure 2.1 for an illustration.
Figure 2.1: An illustration of a synchronous call. At 1, object A has some method invoked by a process before being preempted by a call from somewhere to some method in object B at 2. At some point in this method at 3, a synchronous call is made to some method in A. Since A is already locked, B can not access A. Therefore, control is given back to A at 4 which finishes executing at 5, where it starts handling B's call as well, and finishes and returns a value back at 6, where B can finish its own execution.

Asynchronous calls

An asynchronous call from Bar to Foo in the same situation will not wait for A to become available; instead, Bar continues executing and the asynchronous call will have the effect of deferring the invocation of Foo to when A is available. This has the penalty of disallowing Bar from using any return value given by Foo. See Figure 2.2 for an illustration.

Figure 2.2: An illustration of an asynchronous call. At 1, object A has some method invoked by a process before being preempted by a call from somewhere to some method in object B at 2. At some point in this method at 3, an asynchronous call is made to some method in A. Since A is already locked, the call can not immediately access A, but being asynchronous, it is deferred while B finishes executing. A resumes executing its method at 4 and the asynchronous call from B can now be handled at 5.

Synchronous and asynchronous calls can be made with the SYNC and ASYNC\(^1\) macros in TinyTimber. These macros take as first parameter the object to lock, as second parameter the method in the object to call, and as third parameter a single 32-bit argument to the method. The ASYNC method also takes two additional values: a baseline offset and a deadline offset (see the next section).

2.1.2 Scheduling

Methods can also be invoked with a time delay. To allow for delayed execution and to facilitate hard real-time tasks, each method call is associated with two time values: a baseline and a deadline.

\(^1\)Actually, it is called SEND but for the remainder of this thesis SEND is redefined to be ASYNC to reduce the confusion.
The baseline of a method call is the point in time at which that method was scheduled to start. The beginning of any chain of method invocations (Foo calls Bar calls ...) always starts with some sort of external stimulus, or interrupt, so the baseline of the first method in the chain — the interrupt handler (Foo) — is set to the point of time that the interrupt happened. If the first method calls another method synchronously, the baseline of the second method is set to equal the baseline of the first method, since synchronous calls are never explicitly delayed. However, methods can also be invoked with a time delay, which can be done by an asynchronous call specified with a baseline offset. Such an invocation will be dispatched when the current time value equals the baseline of the calling method plus the specified offset. The baseline of the second method will then be the baseline of the first method plus this offset.

The deadline of a call indicates the amount of time within which the method must finish its execution. It is specified relative to the baseline of the method, and serves to prioritize method execution if there are several calls that are executing simultaneously. If there are several concurrently executing tasks, the method with the earliest deadline is executed.

2.1.3 Some implications
These constructs allow a wide variety of different calling patterns to be implemented. One of the most important patterns is the concept of a cyclic task, which repeatedly performs some sort of operation with some given period. This can be implemented by an object with a method that contains an asynchronous call to itself with the given period as baseline offset.

Something to be avoided is the notion of a deadlock, which can happen if there are two different objects A and B where A calls some method in B synchronously, which in turns calls some method in A synchronously. During the synchronous call to B, the calling object A will wait for B to return control, which will never happen because B will issue a synchronous call which will wait for A to finish. These two objects are then in a state of deadlock. This extends to any chain of synchronous calls that contains a loop, and the way to avoid this from happening is to turn one of the synchronous calls in the loop to an asynchronous call.

The general mechanisms of reactive objects described so far will be part of the design of the virtual machine described in this thesis and proper support will be designed into its instruction set in order to facilitate the concurrent and timed tasks that are present in many embedded and real-time systems.

2.2 Virtual machines
A thorough definition of the concept of a virtual machine can be quite extensive, so within the context of this thesis, a virtual machine is an emulated system environment that provides the computational resources necessary to perform programmed tasks [10]. Many virtual machines consist of some type of memory along with machinery that reads and writes that memory based on the contents within, just like most real computer architectures. Any practical virtual machine is also equipped with means to access processes and I/O devices of the host system to leave meaningful side-effects. Consider for instance a Java program reading from the keyboard and writing information to the computer screen; such a program runs on a virtual machine, but still accesses routines on the host system in order to communicate with the user.
Many of the commonly used virtual machines such as the Java VM or the Lua VM run bytecode, which is code consisting of instructions written in an instruction set designed for efficient execution by a software interpreter. Such an interpreter reads and executes each instruction in turn by performing the requested state change inside the virtual machine. These instructions often mimic the instructions of a real computer system or CPU, such as writing to some register or memory location or conditionally branching to other parts of the program. As an alternative to direct interpretation, bytecode can be compiled, either in advance or as the bytecode is executed, into the native instruction set of the host machine. This enables the real world CPU to execute the code directly which allows for much faster execution.

The architecture of a virtual machine together with its instruction set usually falls into two distinct categories: stack-based or register-based [11]. In a register-based architecture, the primitive operations, like mathematical operations, logical operations, branching, and similar operate on registers, which can simply be regarded as named memory locations. This is similar to how most practical CPUs operate. The instruction for adding two integers in such an architecture might take the form:

```
add a, b, c
```

which assigns to the register c the sum of the values in registers a and b.

In a stack-based architecture, the instructions are instead carried out in the context of a stack data structure. The above operation might be carried out like:

```
push a
push b
add
```

which leaves the stack with the sum of the values of a and b on top.

The relative advantages and disadvantages of register-based versus stack-based machines have been researched thoroughly [11]. The general results relevant to this thesis suggest that stack-based architectures lend themselves better for easy translation of expressions in higher level languages to instructions [12], whereas register-based machines have the potential for generating more optimized bytecode [13].

### 2.3 Compiler construction

A compiler is a program that can read a program in a source language and translate it to a semantically equivalent program in a target language. Strictly speaking, the source and target languages can have any form, but for the purposes of this section it is assumed that the source language is some form of source code and the target language is a set of low level instructions for some target machine.
The general process of compiling a program can be conceptually partitioned into a number of different phases carried out in sequence [16] as depicted in Figure 2.3. As illustration, the following single line of input source code will be followed through the different phases of a compiler in the next few subsections:

```
var = 3 + 4
```

2.3.1 Lexing

First, the source code is analysed lexically and is broken down and categorized into discrete lexemes, which are the strings of text that match the fundamental tokens of the language. These tokens can include include identifiers, operators, numbers, and so on. For the tiny working example given above, the code would be transformed into a stream of tokens

```
var = 3 + 4
```

Each token is associated with some classification information, like 3 is a number, + is an operator, and so on. Just as identifying the word class (such as noun, verb, etc) of a sentence in a natural language helps us parse the sentence, this classification of lexemes paves the way for parsing an expression in a programming language, which is done in the next phase.

2.3.2 Parsing

Source code written in some programming language is very much like a sentence in a spoken language in that the source code must follow the set of grammatical rules specified by the language. Just as one can parse a sentence like "I love sheep" into a verb phrase consisting of a subject, a verb and an object, the code \( \text{var} = 3 + 4 \) can be parsed into an assignment statement consisting of a target value, the assignment operator and an expression.

The grammar of a computer language is commonly expressed in the Backus-Naur Form (BNF) notation. The relevant part of such a grammar to the example is shown in Listing 2.4, which expresses an assignment in terms of a variable, an equals sign and an expression, and so on. Each line is a grammatical rule consisting of several alternatives separated by a vertical
bar. For example, an expression could either be a plus expression, a minus expression or an atom, the latter which in turn is either a variable or a number.

```
<assignment> ::= <var> "=" <expr>

<expr> ::= <plusexpr> | <minusexpr> | <atom>

<plusexpr> ::= <atom> "+" <expr>

<minusexpr> ::= <atom> "." <expr>

<atom> ::= <var> | <num>
```

Listing 2.4: Example of grammar using the BNF notation. Here it is assumed that var and num are tokens produced by the lexer.

The grammatical structure of a particular program can be visualized in a parse tree, such as in Figure 2.5.

```
<assignment>
  <var>
    var
  
  =

<expr>
  <plusexpr>
    <atom>
      3
    +
    <atom>
      4
```

Figure 2.5: The parse tree for the expression var = 3 + 4 from the grammar in Listing 2.4.

The process of manually writing a parser that turns source code into a parse tree can be quite involved, especially for large and complicated grammars. In practice, a parser generator such as Yacc, Bison or ANTLR is used that transforms a grammar like the one shown in Listing 2.4 into a code module that takes a source program as input and gives a data structure representing a parse tree as output. This module can then be used as a component in a compiler.

### 2.3.3 Code generation

The parse tree generated by the parser carries with it all relevant syntactical information about the source code, allowing for an analysis of the semantics of the program. One can walk through the parse tree and perform type checking, create data structures to track variables and their scopes, and similar.
After the necessary analyses, the compiler can use the parse tree together with additional information gathered along the analysis steps to generate target code. This can often be done by traversing the parse tree recursively in a depth-first fashion, generating some code at each node. The result could for example look like Listing 3.1.

2.3.4 Intermediate code and other steps

In real-world compilers, the code generation step typically first produces intermediary code, which is a simple language that lends itself well to further optimization. This intermediary code can be analyzed and further broken down into various forms and data structures that extract more information about the structure of the program, such as loops and branches. This kind of analysis allows the compiler to perform various optimizations, which results in more efficient intermediary code that can be further translated into real machine instructions.

This intermediary representation also serves to decouple the front end of the compiler (lexing, parsing and intermediary code generation) from the back end of the compiler (code generation). Once code has gone through the front end, the rest of the compiler does not need to know what the source language was, which means translating $N$ different source languages to a target language only requires changing the front end of a given compiler. The same goes with the back end of the compiler: to translate a source language to $M$ different target languages, only the back end part needs to be changed. All in all, to translate $N$ source languages to $M$ target languages, this only requires the implementation of $M + N$ different front ends and back ends, instead of $MN$ different compilers.
3 | Design

Keeping the overall scope of the thesis in perspective, this chapter defines the programming standard that allows for hardware-agnostic programming and reprogramming of a system as described in Section 1.2. In section 3.1 the computing environment of the virtual machine is described. Section 3.2 describes the protocol used to transmit a program to the virtual machine, and in Section 3.3 a high level language based on the instruction set of the virtual machine is described.

3.1 Virtual machine design

The virtual machine is an emulated computer architecture that serves as an abstraction layer between the outside developer and the actual hardware running on the system. This section explains the overall architecture of the virtual machine, together with the instruction set which is used to program the virtual machine to perform actions. The required structure of such a program is also illustrated, and a few necessary programming conventions are explained in some detail.

3.1.1 Execution environment overview

Since compiler optimization is beyond the scope of this thesis, and in order to simplify the construction of the compiler by avoiding problems like register allocation, a stack-based architecture is adopted for the virtual machine. This approach involves having part of the memory available for the virtual machine serve as stack space, where the arguments for the various instructions are pushed onto, and the results thereof are popped from. For instance, the sequence of instructions shown in Listing 3.1 yields the sequence of stack configurations shown in figure 3.2.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>push byte 3</td>
</tr>
<tr>
<td>2</td>
<td>push byte 4</td>
</tr>
<tr>
<td>3</td>
<td>add byte</td>
</tr>
<tr>
<td>4</td>
<td>pop byte [var]</td>
</tr>
</tbody>
</table>

Listing 3.1: Some operations that manipulate the stack.

As illustrated in figure 3.2, the stack grows from high memory addresses to lower addresses. This convention carries the benefit that the stack pointer always points to the top element of the stack (rather than the first free element past it). Apart from the stack memory itself, a few registers are required to bookkeep information about the configuration of the stack: Inherently, one register is needed to keep track of the top of the stack, which is commonly called the stack pointer, here denoted $sp$. Also, in order to facilitate a classic function execution frame (see Section 3.1.5) with local variables and function arguments, a frame pointer comes in handy, which is referred to as $fp$. The instructions themselves are
read from the memory location pointed to by the program counter $pc$ which is updated after each instruction has been executed.

To enable preemption and concurrent execution, the instruction set contains instructions to facilitate a TinyTimber model of concurrency. When a new task is started — that is, a method is called asynchronously — a new thread consisting of a stack and the three registers above is instantiated for use for that task. After the task has finished, the thread is recycled and ready for other tasks. In addition to the memory used for the different threads, there is a shared memory area which contains static variables such as objects and their data fields, as well as program code.

An illustration of the conceptual layout of the virtual machine is given in figure 3.3.

![Figure 3.3: An overview of the computing environment of the virtual machine.](image)

In practice, the memory has a unified address space (a von Neumann architecture), so any part of the memory can be read from, written to or executed.

### 3.1.2 Instruction set overview

Memory limitations are a typical constraining factor when designing programs for embedded systems [17], so every relevant operation, like arithmetic and logical operations, can act on
word sizes of either 8, 16 or 32 bits. This allows the programmer to conserve memory by choosing the smallest data size necessary for some variable, which is beneficial especially when declaring large arrays of data. Memory addresses are 16 bits long, and words larger than 8 bits are laid out in little-endian format.

The binary encoding of each instruction consists of a number of bytes (between 1 and 5 depending on the instruction), where the first byte in the instruction is the opcode of the instruction and the rest of the octets serve as the operand for the instruction, such as a memory address or a number constant. This uniform structure allows the virtual machine to quickly determine how to decode and execute each instruction.

A full list of the instructions executable by the virtual machine is given in Appendix C; here follows a summary.

- **Push/pop type instructions.** The push instructions serve to populate the stack with arguments for the other types of instructions. Bytes, words and double words can be pushed from named memory addresses, other positions in the stack, or given directly in the instruction as immediate values. The pop instructions transfer the results from the stack back into memory or into the stack itself. The stack pointer can also be manually increased or decreased.

- **Operational instructions.** These instructions include the most common arithmetic operations, logical operations, and shift operations. Also some comparison instructions that may serve as a preamble to the branching instructions are included here. These latter ones, like `sez` or `sgez`, compare the top of the stack with zero and swap the top with a byte of either a 0 or 1 value depending on the instruction.

- **Subroutine instructions.** The `call`, `sync`, and `async` instructions are included here which serve as the means by which to directly, synchronously, and asynchronously invoke methods. These latter instructions require a number of arguments to be pushed onto the stack before issuing the instruction.

- **Branching instructions.** This group consists of the `jmp` instruction which jumps to a labeled point in the code unconditionally, and the `jez` or `jnez` instructions which pop the top byte off the stack and branch to the given address if the byte was either zero or non-zero respectively.

### 3.1.3 The anatomy of a program

Every program executable by the virtual machine is conceptually and sequentially divided into three segments: the data segment, the code segment, and the extern segment. The data segment holds all statically defined memory such as class variables and global variables. The code segment contains all executable instructions of the program. The extern segment is composed of strings that correspond to names of existing functions predefined by the virtual machine that can be called (Section 3.1.5). Any reference to a string within this last segment is linked to the actual function corresponding to the string when the program is loaded. As illustration of the above, and as an introduction to the language used to write low-level code, 3.4 shows a tiny program that does nothing but toggle an LED light when it starts. This assumes
that there is a function defined and implemented by the virtual machine on the system called
`toggleLed`.

```
.entry
Main:
dword 0
.code
.entry
Main_main:
call toggleLed
ret 0
.extern
toggleLed:
  "toggleLed"
```

*Listing 3.4:* Tiny program showing the basic structure of a program executable by the virtual machine.

The first line defines the entry object, which is the object that gets called (and mutex locked) upon entry of the program. For reference, it is labeled `Main` and contains a required single `dword` (double word of 32 bits), which is initialized to zero and contains room for synchronization information used by the virtual machine. After this, the code segment is defined, which contains the entry method labeled `Main_main`. This method simply calls the function `toggleLed`, before the method returns. The extern segment comes last and contains the string "`toggleLed`" which is referred to by the label with the same name.

### 3.1.4 Loading a program — the binary format

The above program is converted by an assembler (see Section 4.3) into its binary format, which is a string of initialized data, instructions, and strings corresponding to each segment of the program. The entire program is also preceded by an 8-byte header which consists of the address to the entry object, the program section, the entry method, and the extern section, respectively. This header is used by the virtual machine to properly load the program after having received it. Every memory address in the program is relative to the beginning of the program, so in Listing 3.4, the `Main` object has address 0, and the extern segment starts at address 10 (0xA), since the `Main` object has size 4 and the two instructions in the main method each have size 3. Labels and directives are not directly compiled to byte code, but are used by the assembler to calculate addresses and header information, respectively.

### 3.1.5 Method invocation techniques, the stack frame

The instruction set allows for four different ways of invoking a method: asynchronously, synchronously, locally, or externally. The local (or classic) way is used when an object calls a method inside itself, which implies that no regards has to be given for concurrent behavior since the object is already locked to calls from other objects. An external function call, as explained earlier, is an invocation of a function outside the user-supplied program that is hard-coded by the virtual machine. Such an external function is intended to serve as the interface between the virtual machine and the physical devices of the system.
To speed up method invocations, there are a few specialized instructions such as \texttt{call} and \texttt{ret}, which take care of some of the execution frame logic when calling and returning from methods. These instructions assume a specific stack frame structure, which is discussed below.

From the point of view of the caller, a method invocation is exactly like the procedure for a general operation (such as addition) shown in Listing 3.1 and Figure 3.2. The arguments to the function are first pushed onto the stack, and then the \texttt{call} instruction is issued, which replaces the arguments on the stack with a return value. The next few subsections will go into detail on how this behavior is achieved.

\textbf{Local method invocation}

First, for simplicity, assume that the method to be called (the callee) does not return a value. Performing a classic method call without any synchronization is then done as follows: First, the arguments to the method are pushed onto the stack, in reverse order, which leaves the callee with the convenience of having the arguments placed on the stack in their formal order and in the direction of increasing memory. After this, the \texttt{call} instruction is issued that does four things:

1. First, it pushes the return address onto the stack. This is the address to the instruction immediately following the call instruction.
2. It saves the current frame pointer by pushing it into the stack.
3. It sets $fp$ to be the top of the stack, which by step 2 contains the previous value of $fp$.
4. Lastly, it sets $pc$ to the address specified by the operand, performing the jump.

The call stack as received by the invoked method is visualized in Figure 3.5. The callee can use the top of the stack and beyond (memory addresses below $fp$) for its local variables, calculations and deeper method invocations. The arguments to the callee are in order at $fp + 4$ and above. The return address is at $fp + 2$, and the old frame pointer is at $fp$.

![Figure 3.5: The typical stack frame used by some function. The dashed part of the stack is the part of the stack used at the behest of the callee whereas the bottom part was prepared by the caller.](image)

When the callee execution reaches the \texttt{ret} instruction, this instruction sets $fp$ to the value pointed by $fp$, which is the $fp$ of the previous frame. It then sets $sp$ to be $fp + 4 + c$, where $c$ is the argument to the \texttt{ret} instruction. That is, the caller can have the top of the stack set to point to any part below the return address. This is to accommodate return values
from functions. If the callee returns a value, the calling function wants to have this value readily available on top of the stack after the callee has finished executing, so a convention is adopted where the callee overwrites the argument portion of the stack with the return value. This means that from the viewpoint of the caller, pushing a number of arguments and then issuing a call instruction results in having those arguments on the stack popped and replaced by the return value of the called function, which is exactly how each logical or mathematical operation works.

If the return value is of larger size than the arguments to the function, the caller needs to make sure there is sufficient space for the return value (or it could overwrite the "return address" and "old $fp" part of the stack). So, for example, if the callee accepts one argument of size 2 and returns a value of size 4, the caller leaves room for the return value by first leaving an empty space on the stack by decreasing $fp with 2 through the push 2 instruction and then pushing the actual argument. After finishing its execution, the callee makes sure that the caller has the return value on top of the stack by issuing a ret 0 statement, which gives control to the caller with the stack pointer pointing to the same location on the stack as the caller left it. This is illustrated in Figure 3.6.

If the return value is smaller than the total size of the arguments, the argument to the ret instruction comes in handy to have the stack pointer point to the return value after the callee has returned control.

![Figure 3.6](image)

*Figure 3.6:* The stack configurations through a function call where the size of the return value is greater than the size of the arguments to the function.

Every function also needs to make sure it leaves enough room for its local variables, so at the beginning of any function, a push x instruction should be issued, where x is the total size of the local variables.
Asynchronous method invocation

When calling a method asynchronously, the async instruction is used. This instruction requires a number of arguments to be present on the stack when issued: First, the method arguments are pushed onto the stack, in reverse order, just as a local function call. After this, the address to the method followed by the address to the object is pushed. Then, a deadline value followed by a baseline value are pushed as dword values of microseconds, and finally a byte value equal to the total size of the arguments is pushed before the actual async instruction is issued.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>push dword 32 -- method argument 2</td>
</tr>
<tr>
<td>2</td>
<td>push byte 0 -- method argument 1</td>
</tr>
<tr>
<td>3</td>
<td>push methodName</td>
</tr>
<tr>
<td>4</td>
<td>push objectName</td>
</tr>
<tr>
<td>5</td>
<td>push dword 1000 -- deadline, 1 ms</td>
</tr>
<tr>
<td>6</td>
<td>push dword 100000 -- baseline, 100 ms</td>
</tr>
<tr>
<td>7</td>
<td>push byte 5 -- total size of method arguments</td>
</tr>
<tr>
<td>8</td>
<td>async</td>
</tr>
</tbody>
</table>

Listing 3.7: The process of setting up and executing an asynchronous call.

Synchronous method invocation

The sync command is used for calling a method synchronously. As usual, it first requires the arguments to the function be pushed onto the stack in reverse order. After this, the method address and the object address are the next arguments to be pushed before issuing the final sync instruction.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>push dword 32 -- method argument 2</td>
</tr>
<tr>
<td>2</td>
<td>push byte 0 -- method argument 1</td>
</tr>
<tr>
<td>3</td>
<td>push methodName</td>
</tr>
<tr>
<td>4</td>
<td>push objectName</td>
</tr>
<tr>
<td>5</td>
<td>sync</td>
</tr>
</tbody>
</table>

Listing 3.8: The process of setting up and executing a synchronous call.

External method invocation

To invoke an external method, the procedure for a local function call is followed, except that the address in the call instruction should be replaced with the address of its corresponding zero-terminated string in the extern section. When the program is uploaded to the virtual machine, this string is used to replace the given address to the actual memory location of the requested function. This type of call was shown in Listing 3.4.

3.2 Communication design

This section explains the communication protocol used for uploading a program to a system hosting the virtual machine discussed above.
3.2.1 The protocol for programming the virtual machine

It is imperative to prevent corruptions from occurring during transmission of a program to a remote system, so the programming process has been designed with an error detection scheme based on the Automated Repeat Query (ARQ) [14] method. This method was chosen because of its relative ease of implementation. Its main drawback of a slower transmission speed because of the timeout periods used is deemed negligible: the speed of uploading a program is not of major importance since reprogramming a system is more of a special occurrence than part of the ongoing steady-state operation of the device. The ARQ protocol can be implemented on any duplex communication media that can transmit serial data, such as Bluetooth, UART, Wifi, etc.

The frame demarcation is loosely based on the demarcation used in the HDLC standard [15]. The beginning and end of each frame is indicated by a frame delimiter octet, 0x7E. If this octet is part of the actual data sent, it is replaced with two bytes: first the escape octet 0x7D followed by 0x7E with the 5th byte inverted. If the escape octet 0x7D is part of the data, it is replaced by an escape octet followed by the escape octet byte with the fifth byte inverted. Inside a frame, the first byte is always used to denote the type of the frame.

The message formats and transmission procedure

When a client initiates the uploading of a program, the first frame to be sent will take the form shown in Figure 3.9a: first in the frame is the 8 bit number identifying the type of the frame, followed by 2 bytes giving the full length of the program. After this follows a number of payload bytes constituting the first part of the program. Last in the frame is a 4 byte checksum value — this is (tentatively) just the sum of all bytes in the the header and the payload.

When this frame is received by the embedded system, a checksum is calculated from the data in the payload and header segments. If this does not match the supplied checksum segment of the frame, the system does not reply, and waits for a retransmission of the faulty frame. If the frame is deemed to be intact and valid, an ack frame is transmitted back. The ack segment, shown in Figure 3.9b consists of the 8-bit header number which indicates it is an ack frame, followed by a two byte sequence value, which is the total amount of valid program.
data the system has received so far. After this is the checksum, which is computed from the rest of the bytes in the message.

When the client has received an ack header, it will then send the next data in a "send more" frame shown in 3.9c with the corresponding header followed by a two byte sequence value, which is the total amount of data that was sent so far (before this frame), followed by an amount of data bytes, followed by the checksum. The system responds with an ack to this frame or does nothing if it is deemed to be corrupt.

This process continues until the full program has been transmitted.

### 3.2.2 Sending application data

To communicate with a program that is currently running on the virtual machine that supports communication over the same media that it can be programmed through, the HDLC-like frame format as above should be used, as long as the initial header is not one of the ones reserved for the programming protocol. Any frame with the initial octet other than those headers will be sent verbatim to the virtual machine.

### 3.3 High level language

While, strictly speaking, what has been described so far is sufficient as a programming standard, writing programs in assembly language is generally a slow and arduous process prone to errors. Because of this, and as a demonstration of the applicability of the standard, it was decided to design a high level language based on the functionality of the virtual machine. This section will go through the features of the language by a series of examples. The implementation of this language is covered in section 4.4 and the formal grammar of the language is shown in Appendix A.

#### 3.3.1 Overview

Since the instruction set contains instructions to facilitate the TinyTimber model of reactive objects, it is natural to make the higher level language object oriented as well, featuring mechanisms for these objects to invoke methods synchronously and asynchronously. These objects can either be singleton objects or objects created from a class template. The benefit of singleton objects is that there is no need for passing and maintaining a reference to the current object (the equivalent of the C++ or Java this pointer) when calling methods in such an object, saving time and memory. The advantage of instantiated objects is that the source code for multiple objects of the same class does not need to be duplicated, but is instead kept in a single class definition.

A first introduction to this language is given in Listing 3.10 which shows a type of producer-consumer program showing some asynchronous and synchronous calls. The latter type of calls are made from main() to the eat() and produce() methods, as well as from the eat() method to the getFood() method. The asynchronous calls in this case include the produce() method which contains such a call to itself after 200 milliseconds and with a 1 millisecond deadline.

Also to be noticed is the data type notation. Here, the type used for class variables is the int type which is 16 bits wide. Every constant must have a suffix which indicates its type; this
is included to help the compiler with the type checking and code generation. For example, the values for deadlines and baselines in asynchronous calls are of type `long`, with 32 bits width, so they are appended with the `l` suffix. The entry point of the program is always the `main()` method in the `Main` class.

```java
Dog rufus;
Dog willie;

class Dog {
    int mealsEaten = 0i;
    void eat() {
        mealsEaten = mealsEaten + FoodFactory.getFood();
        after 1l sec eat();
        return;
    }
}

object FoodFactory {
    int meals = 0i;
    void produce() {
        meals = meals + 1i;
        after 200l msec before 2l msec produce();
        return;
    }
    int getFood() {
        if(meals > 0i) {
            meals = meals - 1i;
            return 1i;
        }
        return 0i;
    }
}

object Main {
    void main() {
        FoodFactory.produce();
        rufus.eat();
        willie.eat();
        return;
    }
}
```

**Listing 3.10:** An introduction to the high level language.

The language also allows for array assignments and accesses, as shown in Listing 3.11. This code snippet also demonstrates how to use an external function. In this case, the function `uartTransmit` is assumed to be defined by the system. The behavior of this program is simply to read the `arr1` array, write it into another array in reverse order and then transmit it.

```java
extern void uartTransmit(char, char[]);

object Main {
    char[3] arr1 = { 1c, 2c, 3c };
    void main() {
        char[3] arr2;
        int i;
    }
}
```
Pointers to arrays are also supported, so instead of the reversing logic in `main()` above, the call `reverse(arr1, arr2, 3c)` could have been made to the method shown in Listing 3.12. Array indexes always have to be of type `int`, so this code also shows another feature applied on the `char` argument: type conversion.

Reactive behavior towards the system exterior to the virtual machine, such as interrupts, is also solved by the use of external functions. For example, to provide the means to react to input through a uart connection, a function `uartSetCallback` might exist which is used to set a callback method by which to process incoming data. See Listing 3.13.

Here, `function (char) -> void` is the notation for a reference to a function that takes a `char` argument and returns `void`, i.e. nothing.
This chapter describes how the hitherto described designs have been realized, in a bottom-up fashion. The hardware platform that served as the embedded system and the developer tools used are described in Section 4.1, followed by a summary of the implementation of the virtual machine in Section 4.2. The implementation of the code generating parts that reside on the client system is described in Section 4.3 and 4.4.

4.1 Hardware equipment and software tools

In order to minimize the amount of labor potentially spent on tasks such as compiling tools, setting up hardware, debugging third-party software, and other similar activities extraneous to the task defined by this thesis work, the decision was made to use as popular and mainstream a hardware platform as possible. This was also done to maximize the amount of available helpful resources in the form of tutorials, documents and other support to quickly get the hardware and development environment up and running. Since TinyTimber had already been ported to various AVR platforms, a hardware platform that used a microprocessor from this family was deemed preferable. Also, a microcontroller with a decent amount of onboard memory was required in order to have ample space for programs and virtual machine memory.

After some searching, the choice fell on the Arduino Mega 2560 microcontroller board constructed around the Atmega2560 microcontroller which sports 8 kilobytes of memory and a 16MHz operating frequency. The board hosts a number of features, including ports for the input/output pins of the microprocessor, and conveniently also a built-in USB-to-serial converter on its USB connection. To program this board, the AVR Dragon programmer was used.

The JTAG interface of the Mega 2560 was enabled via the ISP programming interface in order to enable on-chip debugging. The Arduino bootloader was erased, yielding a bare-metal AVR platform. JTAG debugging turned out to be very handy during the development process, allowing any running program to be single-stepped through either line-by-line or by the use of breakpoints. The internal state such as values of variables could also be inspected while paused.

As developer environment, Atmel Studio was used, which is a port of Visual Studio created for development of Atmel microprocessor platforms.

TinyTimber was easily ported to the Atmega2560 since the latter is quite similar to the AVR microcontrollers already supported by TinyTimber. Only a few timing constants needed to be updated to accommodate the different clock rate, and a couple of interrupt vectors had to be renamed.
4.2 The virtual machine

The virtual machine was implemented in C, specifically using the avr-gcc compiler used by Atmel Studio. Since the virtual machine together with its instruction set was designed to facilitate a TinyTimber like model of concurrency, it made sense to also use the existing implementation of TinyTimber as the basis of the implementation of the virtual machine. The alternative of constructing an entirely new separate concurrency and timing engine for the relevant parts was deemed not to have any considerable benefits, especially with the substantially added development effort in mind.

4.2.1 Linking and loading a program

After having fully received a program over the communication link, the virtual machine first retrieves the information about the entry points and segment positions in the header portion of the program message. After this, every instruction of the code segment is iterated through, and any instruction with a memory address operand gets the address (which is relative to the start of the program) relocated to its actual absolute position in memory. Any call instruction that refers to a string inside the extern segment gets the address operand replaced with the address of the actual function referred to, and is also changed into a callee instruction to distinguish it from an internal method call. Then, the extern segment together with the remainder of all memory assigned to the virtual machine memory is assigned to a fixed number of available idle threads. The entry method on the entry object is then launched using the exec function, described next.

4.2.2 Instruction dispatch — the exec function

At the heart of the implementation of the virtual machine is an instruction dispatch loop which continuously reads, decodes, and executes instructions. The dispatcher first reads the value of the $pc register of the current thread. This register points to the first byte of the current instruction, which contains the opcode of the instruction, so a switch statement is performed on that value to jump to the routine that executes the code corresponding to the instruction. After the instruction has been executed, the $pc register is incremented to point to the next instruction, and the loop is continued anew.

This loop takes place in a function called exec, whose job is to execute the instructions that reside in a method of some object in the bytecode. This function is declared as a standard TinyTimber-callable method, with the first parameter being the object that the method resides in. The second argument to exec is a pointer to an instance of a "argument bin" data structure called VmArgBin. This structure serves to overcome the single-argument limitation that TinyTimber suffers from, and contains all the information necessary to start a new thread on an invoked method. This information includes the arguments to the method, the address to the method and the return address.

---

1 The switch statement is compiled to a table lookup followed by a jump by GCC, so there is no need to manually construct a jump table.
2 However, exec will still be called a function to help separate the code that implements the virtual machine from the bytecode that it executes.
The **sync** and **async** instructions are handled quite differently from most other instructions, since they achieve the concurrency control and scheduling capabilities provided by TinyTimber. They are also the only instructions that give rise to new invocations of **exec**. These instructions are implemented with the help of the **SYNC** and **ASYNC** functions of TinyTimber.

**Handling sync instructions**

Like every instruction with arguments, the execution of the **sync** instruction starts with popping its arguments off the stack of the current thread, in this case the target object and method address. The invoked method address together with the pointer to the current thread object are saved into the **VmArgBin** structure that was passed into the current call of the **exec** function. The formal arguments to the method that are now on top of the stack (e.g. the first two values pushed in Listing 3.7) are left as they were. Then, the return address (the address to the instruction after the **sync** instruction, equal to \$pc + 1) is pushed onto the stack, followed by a dummy "old \$fp" pointer of 0, used to indicate to the callee that this was a **sync** call to the method rather than a local call. These steps have provided the new method with a proper stack frame. Note that this is all done on the same thread object that **exec** is already running on; no new thread is needed for a **sync** statement. To save the actual current frame pointer, it is stored into a local variable. Finally, the frame pointer of the current thread is set to point to the top of the stack (where the "old \$fp" pointer is stored), and an actual TinyTimber **SYNC** call is made to the **exec** function, with the target object as first parameter and the **VmArgBin** as second parameter. This provides concurrency control: if the virtual machine is already executing bytecode on the target object, it is doing so within the context of a TinyTimber **SYNC** or **ASYNC** call to **exec** on that object, which means it is locked and execution has to wait until it has finished.

When **exec** starts on the new object, the function sees that a thread object was provided, so it simply continues using that thread (which has the stack frame readily set up), and only has to set the \$pc register to the method address passed in the **VmArgBin** before it starts the instruction loop. From this point on, instructions are executed as normal, including possible further **sync** and **async** calls that can be made to other objects, but eventually the end of the originally called method will be reached which is embodied by the **ret** instruction. This instruction restores the old \$fp from the stack, and sets \$pc to be the return value pushed on the stack. Now **ret** has to figure out if this method was called synchronously, asynchronously or locally in order to determine what to do next. First, it checks the value of \$fp. Since this is 0, **ret** knows it was not a local call, so it needs to return control from the **exec** function. Since a real return address was given, it knows this is a **sync** call, and simply returns from the **exec** function. Back at the **exec** function of the caller object, the thread finds itself just after the TinyTimber **SYNC** call of the dispatcher of the **sync** instruction. The value of the old \$fp that was saved in a local variable can now be restored, and the next instruction can be executed normally.

**Handling async instructions**

The implementation of the **async** instruction is similar to the above, except together with the target object and method address, the dispatcher needs to pop information about the baseline
offset and deadline as well as the total argument size off the stack. A new VmArgBin structure is retrieved and filled with the necessary information, and the argument size is used to copy the part of the stack that contains the method arguments into a buffer in the argument bin. The target method address is copied as well. After this, an ASYNC call is made to exec, with the provided method address as first argument, and the baseline offset and deadline as arguments to ASYNC. This provides the scheduling functionality, since the bytecode of the target method will only start executing after the given offset. When exec starts on the new object, it sees that no thread object was given, so it fetches a new thread object, initializes the registers to their appropriate values and pushes the arguments provided by the argument bin onto the stack, followed by null values for the "old $fp" and return address parts of the argument stack. Then, the instruction dispatch loop is begun, which loops normally. When the ret instruction is reached, it sees that both the "old $fp" and return addresses are set to zero on the stack, so before returning, it recycles the argument bin and the thread object for others to use.

As an illustration, consider the program in Listing 4.1. Here, a method in object A asynchronously calls a method in object B with some delay. The latter method calls yet another method in object A synchronously. This behavior is illustrated in Figure 4.2, and Figure 4.3 shows how the actual calls are set up to the exec functions that execute the bytecode within.

```
-- data segment
ObjectA:
dword 0
ObjectB:
dword 0
-- part of the code segment below
ObjectA_methodA:
  push ObjectB_methodC
  push ObjectB
  push dword 1000000
  push dword 0
  push byte 0
  async
  ret 0
ObjectA_methodB:
  -- do stuff
  ret 0
ObjectB_methodC:
  push ObjectA_methodB
  push ObjectA
  sync
  ret 0
```

Listing 4.1: Partial source code showing some calls between two objects.
4.2.3 External functions

After the program has been linked, the operand to the `call` instruction is assumed to be an address to a function that takes a single operand: a pointer to the thread object. It is then up to that function to manually pop arguments off the stack and push a result back onto the stack after having performed the purpose of the function. It is also up to the function to impose thread-safety; if the function is invoked by some task A, and then during its execution it is preempted by task B that calls the function, it is the responsibility of the function to make sure no race conditions occur. This can easily be solved by having the function perform a `SYNC` call to some object.

4.2.4 Error handling

Any type of error handling was decided to be beyond the scope of this thesis, so there is no out-of-bound checking for address operands to memory access or jump instructions, or other potentially system breaking occurrences.

4.3 The assembler

The purpose of the assembler is to turn a file consisting of instruction mnemonics into the serialized array of bytes used to program the virtual machine. This is performed in three phases: first, the assembler parses each line of the source code into a data structure that contains the equivalent bytecode of the instruction, or in the case of labels or directives, information about its position. Any label is inserted into a map between labels and memory addresses. Some instructions might contain forward-references to labels that have not yet been defined at that point in the program. Such instructions have their operands set to zero, and are inserted into an errata, which maps instructions to labels. When the entire program has been parsed (and all labels have been defined) the second phase consists of going through the errata and assigning the contained instructions their proper operands based on the label table. Finally, in the
third phase, all the bytecode of the statements is added together to form the final program, with the information given by the entry point and segments prepended as the header.

4.4 The compiler

The compiler was written in Java, and ANTLR [18] was chosen as the parser generator. ANTLR takes as input a grammar that specifies a language and can from that grammar generate source code for recognizers, parse trees and walkers that can traverse these parse trees. This latter feature was used for the code generation part of the compiler.

4.4.1 Parsing

The full grammar for the language is shown in Appendix A. Note that, as it stands, this grammar is inherently ambiguous (e.g., the expression subrule), but ANTLR resolves ambiguities by selecting the highest matched alternative for a rule. For example, the expression $4+3\cdot2$ will be interpreted as $4+(3\cdot2)$ and not $(4+3)\cdot2$ since the alternative expression+expression is listed higher than the alternative expression*expression in the alternative list.

In the compiler, the ANTLR-generated parser is used to transform an input program into a parse tree whose nodes are objects inherited from a base class called ParserRuleContext. Each such object has a name on the form ruleContext, where rule is the name of the grammatical rule matching that node. For example, if the program contains an expression $1+3$, the parse subtree that ANTLR generates for this expression, given the grammar in Appendix A, consists of an AddExpContext node that has two NumExpContext children. These nodes contain all relevant information about the part of the input program that matched the rule, such as the matched string, the matched subrule nodes, and more.

ANTLR was also used to generate a (base) class that contains a visitRule method for every rule in the grammar. By extending this class and overloading its methods, all the code generation logic is contained in this derived visitor class. The base class also has a templated return value of the visitRule methods, which was bound to be of an object of a class inheriting a Type class. This is used for type checking.

4.4.2 Symbol table construction

Before being able to translate arbitrary lines of high-level code into their corresponding sequence of low-level instructions, some preparatory work has to be undertaken.

To resolve identifiers, or symbols, such as variable and method names, a symbol table needs to be constructed to keep track of all symbols that are accessible from any given part of the code (or equivalently, any given part of the parse tree). Part of this symbol table is constructed by a first partial perusal of the parse tree: From the top program rule, the nodes for all class declarations are visited to build a symbol table of the names of the different classes and their contained methods. The external function declarations and class object instantiations are also added.

The symbol table is ordered hierarchically with class names and external functions put into a toposmost global scope, and class names and variables put into class scopes that have the global scope as parents (see Figure 4.4).
As the nodes of the parse tree are perused in the code generation step, a pointer to the closest scope is maintained. For example, when a method node is visited, a method scope is constructed with the given formal arguments and local variable declarations, and this can be used in conjunction with the parent scope to resolve any identifier given — if some symbol was not found in the current scope, the parent scopes are searched. After the visitor exits the method node, the scope pointer is assigned to point to the parent of the current node, which is the scope of the currently visited class.

4.4.3 Code generation

Any sort of program optimization is outside the scope of this thesis, so no transformations or additional intermediary representations of the program are carried out before the code generation step. Instead, code is generated directly in a single depth-first pass through the parse tree, using the symbol table generated earlier. The general method will be explained for the most relevant nodes of the parse tree in the next few subsections.

In the pseudocode segments given below there are a few instances of notation used. gen(expr) is the generated code for the expression expr, size(var) is the size of the variable var, which is either 1, 2 or 4, and strsize(var) is the size string of the variable, which is either byte, word or dword. The notation label(x) stands for the label associated with the entity x. The function addr(var) maps a variable var into its address, which can be indicated by either a label or a position on the stack.

Generating code for expressions

Expressions are particularly easy to generate code for. Through an inductive argument, it can be shown that the result of any expression is a single value on the stack. Because of this, an entire expression tree can be treated as a single operand when generating code for any expression or statement. Hence, for some general node in an expression tree, code is first generated for one subtree, then the other subtree and finally for the operation corresponding to the node. Type checking can be done at the same time as well.

---

3 This can be shown by strong induction on the depth of an expression tree. Any tree with depth 0 can only be a direct value, which is generated by pushing it onto the stack. Assume any tree of depth less than or equal to k evaluates to a single value on the stack. A tree of depth k + 1 must consist of a root node and up to two subtrees of depth less than or equal to k. These two subtrees each evaluate to a single value on the stack by the induction assumption. Any operation, and in particular the one represented by the root node, transforms these two values to a single value on the stack, which means a tree of depth k + 1 evaluates to a single value as well, and hence trees of all depths by induction.
public Type visitLogOrExp(GravelParser.LogOrExpContext ctx) {
    Type a = visit(ctx.expression(1));
    Type b = visit(ctx.expression(0));
    if (!a.equals(b) || !(a instanceof BoolType))
        return reportError(ctx, "Both arguments to || must be of type bool");
    emitProgramString("or byte");
    return new BoolType();
}

Listing 4.5: An excerpt from the compiler.

The suitability of the stack-based machine when it comes to generating code for expressions is illustrated in the code snippet given in Listing 4.5, which shows the visitor method for the grammatical rule denoted by #LogOrExp in Appendix A.

In the example, code is first generated for the subtrees of the or expression node, which are expressions themselves. Then, rudimentary type checking is performed, before the final or instruction is generated. A BoolType value is then returned, which indicates that this expression evaluates to a boolean type. This type information is used by the parent node which might be a function call or another boolean expression.

The code generation for this node, and any arbitrary binary expression \( \text{leftExpr} \oplus \text{rightExpr} \) can be written in a general form as expressed in Listing 4.6.

```
1 gen(rightExpr)
2 gen(leftExpr)
3 gen(⊕)
```

Listing 4.6: Code generation for a general binary expression \( \oplus \).

For the or expression shown in Listing 4.5 above, \( \text{gen(⊕)} \) is or byte.

**Method bodies**

All executable code resides within method bodies. At the entrance of any given method body, space is reserved for the local variables on the stack (See Figure 3.5) by issuing a push \( x \) statement where \( x \) is the total size of the local variables. Then, code is generated separately for each statement in the method body in turn. At the end of the method body, the stack space is restored with a pop \( x \) instruction, before a ret instruction is issued.

**Method calls**

Local method calls method(\( a_1, a_2, \ldots, a_n \)), where \( a_1, a_2, \ldots, a_n \) are expressions, are also straightforward to generate code for. First, in case the return value is bigger than the method arguments, space is reserved for this value on the stack by generating a push \( x \) instruction, where \( x \) is the difference between the sizes of the return value and the method arguments (see Section 3.1.5). Then, code is generated for each argument expression, from the rightmost argument to the leftmost as in Listing 4.7.
If the object that contains the method is an instance of a class, the pointer to that object is pushed onto the stack as an invisible first argument to the method. This pointer can be retrieved from global scope if the target object is mentioned explicitly in the form of \texttt{obj.method()}. If the method call is just a local call on the form of \texttt{method()}, the pointer is accessible as the invisible first argument to the current method at $\_fp + 4$.

For \texttt{sync} calls (or, calls to other objects), the required method address and object pointer parameters are pushed as well as the actual \texttt{sync} instruction. For asynchronous calls, code is created in much the same way as for synchronous ones, except the additional timing information as well as argument size are pushed as well.

### Assignments

Code generation for assignments is based on the variant of the \texttt{pop} instruction which takes two arguments: an address on top of the stack and a value to assign to that memory address below this. The code for an assignment is thus generated by first generating the code for the right hand side expression of the assignment, and then resolving the identifier on the left side and pushing the appropriate address for that identifier, followed by the \texttt{pop} instruction. In other words, the statement \texttt{var = expr} is generated as in Listing 4.8.

```plaintext
1 gen(expr)
2 push addr(var)
3 pop strsize(var)
```

Listing 4.8: Code generation for an assignment.

### Array lookups and assignments

To access some value given by an array expression such as \texttt{arr[expr]}, code is first generated for the expression that yields the index of the array, which must be of type \texttt{int}. This is followed by a multiplication of this index by the element size of the array, followed by an addition with the starting address of the array. The value at that address is then pushed to the stack with a \texttt{push} instruction. This process is shown in Listing 4.9.

```plaintext
1 gen(expr)
2 push word size(arr)
3 mul word
4 push addr(arr)
5 add word
6 push strsize(arr)
```

Listing 4.9: Code generation for an array lookup and assignment.
Listing 4.9: Code generation for an array lookup.

Array assignments like \( arr[expr_1] = expr_2 \) are performed similarly, except code for the value to be assigned is generated first, and the final push is replaced with pop instruction as illustrated in Listing 4.10.

```assembly
1 gen(expr_2)
2 gen(expr_1)
3 push word size(arr)
4 mul word
5 push addr(arr)
6 add word
7 pop strsize(expr_2)
```

Listing 4.10: Code generation for an array assignment.

Control statements

The if statement on the form \( \text{if}(expr) \; stmt \) first generates code for the conditional expression, then evaluates the falsehood of the result of that expression with the jez instruction, which is used to jump to a label beyond the statement block. Between the jez instruction and the given label, the consequent statement is generated. This is shown in Listing 4.11.

```assembly
1 gen(expr)
2 jez lbl
3 gen(stmt)
4 lbl:
```

Listing 4.11: Code generation for an if statement.

Code generation for the while(\( expr \)) \( stmt \); statement involves first generating a label to jump back to after each iteration. This is followed by the generated code for the conditional expression, and a jez statement that jumps beyond the entire while loop if the conditional expression is false. After this comes the code for the loop statement, followed by an unconditional jump back to the first label. The process for generating the code for a while statement is summarized in Listing 4.12.

```assembly
1 lbl1:
2   gen(expr)
3   jez lbl2
4   gen(stmt)
5   jmp lbl1
6 lbl2:
```

Listing 4.12: Code generation for a while statement.
5  | Evaluation

The most relevant part of the thesis work to evaluate is the usability and performance of the computational environment offered by the virtual machine. This is because the other major part, the high-level language, is in a sense arbitrary — many other different types of languages can be written for the same instruction set while still following the programming interface exposed by the virtual machine.

5.1 Performance

As noted earlier in the case of the Darjeeling virtual machine, the main drawback of running an interpreted virtual machine can be argued to be the great execution overhead in fetching, decoding and executing every instruction in software. It is therefore pertinent to measure how big this performance penalty is.

To get an estimate of the performance overhead, the amount of CPU cycles that each instruction requires to execute was measured for a number of instructions. For this purpose, the timer functionality of the AVR chipset was used to count the number of cycles used by a single iteration of the instruction dispatch loop. The cycle counter was read before and after a single instruction dispatch and the difference of these values was then sent through the serial connection to the development computer.

Currently, there is no runtime error checking implemented in the virtual machine, but in order to get an idea of the performance penalty of adding this, the process described in the previous paragraph was repeated after adding such checks to the few given instructions. With this enabled, the push instructions only execute if the stack will not overflow, the jmp and pop instructions make sure that the operand is within the memory assigned to the virtual machine, and the instructions that require operands on the stack also ensure that the stack will not underflow. The result is summarized in Table 5.1.

<table>
<thead>
<tr>
<th></th>
<th>push dword num</th>
<th>push [fp]</th>
<th>pop [fp]</th>
<th>sub</th>
<th>add</th>
<th>sgz</th>
<th>jez</th>
<th>jmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycles (no checks)</td>
<td>90</td>
<td>103</td>
<td>102</td>
<td>101</td>
<td>103</td>
<td>104</td>
<td>77</td>
<td>56</td>
</tr>
<tr>
<td>cycles (checks)</td>
<td>101</td>
<td>118</td>
<td>108</td>
<td>115</td>
<td>115</td>
<td>122</td>
<td>92</td>
<td>66</td>
</tr>
</tbody>
</table>

Table 5.1: Cycle costs for some selected instructions without and with run-time checks. Omitted sizes are of type dword.

To verify that this correctly measured the cost of each instruction, the program in listing 5.2 was used. When it receives any input over the uart connection, it runs the test() function before sending dummy data back over the serial channel. This allows an outside observer to approximately measure how long the loop inside the function takes to execute by measuring
the time the machine takes to respond to a transmitted serial message\(^1\). Here, the version without runtime checks is tested.

```c
extern void setUartCallback(char[], function (char) -> void);
extern void uartTransmit(char, char[]);

object Main {
    char[100] input;
    void test() {
        long i;
        i = 0L;
        while(i < 1000000L)
            i = i + 1L;
        return;
    }
    void handleInput(char length) {
        test();
        uartTransmit(2c, "d");
        return;
    }
    void main() {
        setUartCallback(input, Main.handleInput);
        return;
    }
}
```

**Listing 5.2:** Code used for performance measurement.

The while loop in Listing 5.2 compiles to the instructions shown in Listing 5.3.

```assembly
label0:
push dword [fp-4]
push dword 1000000
sub dword
sgz dword
jez label1
push dword 1
push dword [fp-4]
add dword
pop dword [fp-4]
jmp label0
label1:
```

**Listing 5.3:** Compiled code.

Given that the system runs at 16MHz and assuming that the cycle costs measured above are accurate, it is straightforward to calculate that one million iterations of the while loop in

\(^1\) Another approach that was considered was to measure the execution times for 8 different loops that used these 8 instructions in different proportions and solve the resulting linear system for the time used by each instruction. However, for any practical program this gives a singular coefficient matrix since there is a linear dependency between the columns (number of instructions executed): in order to not underflow or overflow the stack, the number of pop instructions must equal the number of push instructions minus the number of operational instructions.
Listing 5.3 would take 58062 milliseconds to execute. The actual measured time was 58329 milliseconds, so the cycle count estimates given earlier can be regarded as being essentially correct. Running the above program with run-time checks enabled gives a calculated execution speed of (exactly) 66000 milliseconds, which constitutes an additional overhead of about 14%.

Since most AVR instructions use between one and three cycles \cite{19}, the virtual machine instruction execution cost of 50-100 cycles can be argued to have an overhead of somewhere between 16x-100x, which, while substantial, compares favorably with the examples of other work given in the introduction of this thesis. Still, such a derivation of the overhead is tenuous given the different nature of the AVR instruction set and the stack-based one devised for this thesis. A more straightforward test was simply to hard-code the above loop in C and measure the time similarly, in which case it finished executing in 2139 milliseconds, giving a 27x overhead with no run-time checks and a 31x overhead with run-time checks enabled.

5.2 Expressibility

While computational performance can be of less importance in certain applications, it is generally advantageous if the computational interface defined is expressable and usable, which is harder and perhaps more arbitrary to define metrics for.

One basic criterion for general usefulness in terms of the computational power of a machine is that of Turing-completeness. A machine fulfilling this criterion means the machine can compute any function whose values can be calculated by an algorithm (Assuming the Church-Turing thesis is true). The property of Turing-completeness holds true for any general purpose programming language and even some cellular automata such as Conway’s game of life \cite{20}, so in any practical sense it does not give any additional information about the practicality of a machine. Still, to show that the machine fulfills this property, a Turing machine has been implemented in Appendix B.

In terms of usability, the implementation of the high-level language itself can be regarded as a favorable evaluation of the practical usefulness of the instruction set of the programming interface. As a demonstration of the applicability of this language in turn, the classic games Snake and Sokoban were easily implemented and used in conjunction with a graphical user interface on a local computer.
6 | Future work

Being a limited body of work in time and scope, a number of important topics have been omitted; hence, there are some natural directions in which this work can be extended. The compiler and high level language are excluded from this list of suggestions since they are, as discussed earlier, to be regarded more of an application of the programming standard itself.

6.1 Error handling

This thesis has not defined or any sort of protocol for error handling after transmission, during linking, or during execution. For example, the transmission of a program that contains an external call to a function not defined by the virtual machine should produce a descriptive error message to let the developer know that the requested functionality is not supported by the machine. This sort of error handling should extend to other similar "static" errors such as uploading a program that does not fit in the memory of the virtual machine, or a program with easily recognized syntactic or semantic errors such as code segments containing unknown opcodes.

There are still errors that are difficult to implement a control mechanism for, such as instructions or functions being invoked with a wrong number of arguments, or mismanagement of the execution frame, among many others. Some errors can be handled by adding bounds checking to the instruction dispatcher to make sure that instructions cannot write to, or read from, memory locations outside the memory space assigned to the virtual machine. This would help confine such errors to inside the virtual machine and potentially avoids having them "brick" the entire system. Such runtime checking could be enabled or disabled by the programmer when uploading the program.

6.2 Querying

In order to write code for some given system, a developer might need to retrieve information about its properties. This might include its computational capacities such as memory size or processor speed, or more importantly the supported external functions that expose the internal hardware. This can easily be implemented by adding query and response frames to the communication protocol: for example, such a query frame might ask for a list of all supported functions, and the response to such a query would be frame or sequence of frames containing a list of all the external functions of the system, along with their arguments.

6.3 Information security

To protect a critical system from an unauthorized client performing a potentially malicious reprogramming of a system, there should also be some method of authorization during the
programming process. This could be implemented by adding password protection to the re-
programming process, perhaps in conjunction with public-key cryptography.

Similarly, the program itself could be transmitted using encryption using some sort of ci-
pher negotiated through the authorization process, similar to how transmission over the web
is encrypted.

### 6.4 Extended virtual machine capabilities

One obvious extension of the functionality of the virtual machine is support for floating point
numbers and operations. This could be realized by extending the instruction set with floating
point versions of all current numeric integer operations, along with instructions to convert
between the already existing data types and floating point values.

### 6.5 Context switch elimination

A slightly more advanced improvement to implement would be upgrading TinyTimber to use
the SRP protocol as mentioned in [9], which would remove the necessity of using different
threads and separate stacks, thus improving memory efficiency and overhead from context
switching.
A | Appendix: Language grammar

The grammar is written in the BNF-like format accepted by ANTLR. Some regex-like symbols are allowed: an asterisk after an expression means "zero or many", a plus means "one or many" and a question mark means "possibly omitted".

```plaintext
grammar Gravel;
program : externDeclaration* classInstanceDeclaration* classDefinition*;
externDeclaration : 'extern' type identifier '(' (type (',' type)*)? ')' ';' type : baseType brackets?;
brackets : ('[' (NUM)? ']') baseType : 'bool' | 'int' | 'char' | 'long' | identifier | 'void' | functionPtr;
functionPtr : 'function' '('. ((type (',' type)*)?) ')' '->' type;
classInstanceDeclaration : identifier identifier ';
classDefinition : classType identifier '{' classVariableDeclaration* methodDefinition* '}'
classType : 'object' | 'class' classVariableDeclaration : type identifier ('=' (scalarInitializer | arrayInitializer))? ';
scalarInitializer : NUM suffix;
arrayInitializer : '{' NUM suffix (',' NUM suffix)* '}' methodDefinition : type identifier '(' (type identifier (',' type identifier)*)? ')'

    '{' methodBody '}'
methodBody : methodVariableDefinition* statement* returnStatement;
methodVariableDefinition : type identifier ';
statement : assignment | ifStatement | whileStatement | returnStatement

    '{' statement* '}'
    | functionCallStatement | asyncStatement;
assignment : lvalue '=' expression ';
ifStatement : 'if' '{' expression ')' statement elseClause?;
elseClause : '{' 'else' statement '};
```
whileStatement : 'while' '(' expression ')' statement ;
asyncStatement : after expression time before expression time functionCall ';' |
                  after expression time functionCall ';' |
                  before expression time functionCall ';' ;
after : 'after';
before : 'before';
time : 'sec' | 'msec' | 'usec';
lvalue : identifier | identifier '[' expression ']';
expression :
            identifier #identifierExp |
            'true' #trueExp |
            'false' #falseExp |
            NUM suffix #numExp |
            string #stringExp |
            '(' baseType ')' expression #castExp |
            expression ('[' expression ']') #arrayLookupExp |
            identifier '.' identifier #indirectionExp |
            functionCall #functionCallExp |
            '(' expression ')' #parExp |
            '!' expression #notExp |
            expression '*' expression #mulExp |
            expression '/' expression #divExp |
            expression '%' expression #modExp |
            expression '+' expression #addExp |
            expression '-' expression #subExp |
            expression '>' expression #gtExp |
            expression '>=' expression #gteExp |
            expression '<=' expression #lteExp |
            expression '<' expression #ltExp |
            expression '==' expression #eqExp |
            expression '!=' expression #neqExp |
            expression '&' expression #bitAndExp |
            expression '^' expression #xorExp |
            expression '|' expression #bitOrExp |
            expression '&&' expression #logAndExp |
            expression '||' expression #logOrExp ;
functionCall : (identifier.?identifier '(' (expression (',' expression)*)? ')') ;
functionCallStatement : functionCall ';' ;
returnStatement : 'return' expression? ';' ;
suffix : identifier ;
identifier : TEXTNUM ;
TEXTNUM : (CHAR)+(NUM|CHAR)* ;
number : NUM ;
string : STRING ;
CHAR : ('a'..'z' | 'A'..'Z' | '_');
NUM : '-'?('0'..'9')+ ;
STRING : '"' ( ~('
'|'') )*? '"';
WS : (' '|'	'|''|'
'|'')+ -> skip ;
MULTICOMMENT : '/*' .*? '*/' -> skip ;
SINGLECOMMENT : '//' ~('' | '
')* -> skip ;
Appendix: Turing machine

For brevity, it is assumed that the tape, transition functions, and state sets are stored externally. The transition function \( \delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\} \) is here provided as three component functions \( \delta_q : Q \times \Gamma \rightarrow Q, \delta_\Gamma : Q \times \Gamma \rightarrow \Gamma \) and \( \delta_D : Q \times \Gamma \rightarrow \{-1, 1\} \) due to language limitations. These are called getState, getSymbol, and getDir in the code, respectively. Also, for practical reasons, the states are assumed to be identified with natural numbers, and the alphabet is assumed to consist of natural numbers as well.

```c
extern int getState(int, int);
extern int getSymbol(int, int);
extern int getDir(int, int);
extern int readTape(int);
extern void writeTape(int, int);
extern int getInitialState();
extern bool isFinalState(int);

object Main
{
    int tapePos = 1;
    int state = 1;

    bool run()
    {
        int symbol;
        int nextState;
        int dir;
        int symbolWrite;

        state = getInitialState();
        while(!isFinalState(state))
        {
            symbol = readTape(tapePos);
            nextState = getState(state, symbol);
            symbolWrite = getSymbol(state, symbol);
            dir = getDir(state, symbol);

            if(nextState == -1)
                return false;

            writeTape(tapePos, symbolWrite);
            tapePos = tapePos + dir;
            state = nextState;
        }
        return true;
    }
}
```
void main()
{
    run();
    return;
}
Appendix: Instruction set

In each entry in the table below, the first value in hexadecimal is the opcode, which is the first identifying byte of the instruction. The value to the right of the opcode is the total size of the instruction, which is followed by the instruction mnemonic itself.

The contractions $imm$ and $lbl$ stand for immediate and label, respectively. These are values that are issued inside the instruction itself, just after the opcode, in little-endian byte order.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Size</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>3</td>
<td>push $fp+c</td>
<td>Pushes the 2 byte value equal to $fp + c onto the stack.</td>
</tr>
<tr>
<td>0x02</td>
<td>3</td>
<td>push imm</td>
<td>Decreases $sp by $imm.</td>
</tr>
<tr>
<td>0x03</td>
<td>3</td>
<td>push $lbl</td>
<td>Pushes the address to $lbl onto the stack.</td>
</tr>
<tr>
<td>0x04</td>
<td>3</td>
<td>push byte [$fp+c]</td>
<td>Pushes the byte which is found at the address pointed to by $fp + c onto the stack.</td>
</tr>
<tr>
<td>0x05</td>
<td>3</td>
<td>push word [$fp+c]</td>
<td>Pushes the word which is found at the address pointed to by $fp + c onto the stack.</td>
</tr>
<tr>
<td>0x06</td>
<td>3</td>
<td>push dword [$fp+c]</td>
<td>Pushes the dword which is found at the address pointed to by $fp + c onto the stack.</td>
</tr>
<tr>
<td>0x07</td>
<td>3</td>
<td>push byte $lb</td>
<td>Pushes the byte which is found at the address pointed to by $lb onto the stack.</td>
</tr>
<tr>
<td>0x08</td>
<td>3</td>
<td>push word $lb</td>
<td>Pushes the word which is found at the address pointed to by $lb onto the stack.</td>
</tr>
<tr>
<td>0x09</td>
<td>3</td>
<td>push dword $lb</td>
<td>Pushes the dword which is found at the address pointed to by $lb onto the stack.</td>
</tr>
<tr>
<td>0x0A</td>
<td>2</td>
<td>push byte imm</td>
<td>Pushes the byte value $imm onto the stack.</td>
</tr>
<tr>
<td>0x0B</td>
<td>3</td>
<td>push word imm</td>
<td>Pushes the word value $imm onto the stack.</td>
</tr>
<tr>
<td>0x0C</td>
<td>5</td>
<td>push dword imm</td>
<td>Pushes the dword value $imm onto the stack.</td>
</tr>
<tr>
<td>0x0D</td>
<td>1</td>
<td>push byte</td>
<td>Replaces the address on top of the stack with the byte value found at the address.</td>
</tr>
<tr>
<td>0x0E</td>
<td>1</td>
<td>push word</td>
<td>Replaces the address on top of the stack with the word value found at the address.</td>
</tr>
</tbody>
</table>
0x0F  1  push dword
Replaces the address on top of the stack with the dword value found at the address.

0x10  3  pop imm
Increases $sp by imm.

0x11  3  pop byte [$fp+c]
Pops the byte from the top of the stack to the location with address $fp + c.

0x12  3  pop word [$fp+c]
Pops the word from the top of the stack to the location with address $fp + c.

0x13  3  pop dword [$fp+c]
Pops the dword from the top of the stack to the location with address $fp + c.

0x14  3  pop byte [lbl]
Pops the byte from the top of the stack to the location with address pointed to by lbl.

0x15  3  pop word [lbl]
Pops the word from the top of the stack to the location with address pointed to by lbl.

0x16  3  pop dword [lbl]
Pops the dword from the top of the stack to the location with address pointed to by lbl.

0x17  1  pop byte
Pops the top word off the stack (addr), pops the byte below to location addr.

0x18  1  pop word
Pops the top word off the stack (addr), pops the word below to location addr.

0x19  1  pop dword
Pops the top word off the stack (addr), pops the dword below to location addr.

0x1A  3  call lbl
Pushes $pc + 3 onto the stack, pushes $fp onto the stack, sets $fp equal to $sp and sets $pc to the address pointed to by lbl.

0x1B  3  ret imm
Sets $sp to $fp + 4 + imm, sets $pc to the value pointed to by $fp + 2, sets $fp to the value pointed to by $fp.

0x1C  1  sync
Pops an address to the object and an address to the method off the stack and performs a synchronous call to that method, locking that object.

0x1D  1  async
Pops the total size of the arguments, the dword baseline value, the dword deadline value, an address to the object and an address to the method, and all the arguments off the stack and schedules an asynchronous call to that method.

0x1F  3  jmp lbl
Sets $pc to the address pointed to by lbl.

0x20  3  jez lbl
Pops the top byte off the stack, and if this byte is zero, sets $pc to the address pointed to by lbl.
Pops the top byte off the stack, and if this byte is not zero, sets $pc to the address pointed to by lbl.

Pops two bytes off the top of the stack and pushes their byte-sized sum onto the stack.

Pops two words off the top of the stack and pushes their word-sized sum onto the stack.

Pops two dwords off the top of the stack and pushes their dword-sized sum onto the stack.

Pops two bytes off the top of the stack, minuend first, and pushes their byte-sized sum onto the stack.

Pops two words off the top of the stack, minuend first, and pushes their word-sized sum onto the stack.

Pops two dwords off the top of the stack, minuend first, and pushes their dword-sized sum onto the stack.

Pops two bytes off the top of the stack and pushes their byte-sized product onto the stack.

Pops two words off the top of the stack and pushes their word-sized product onto the stack.

Pops two dwords off the top of the stack and pushes their dword-sized product onto the stack.

Pops two bytes off the top of the stack, numerator first, and pushes their byte-sized quotient onto the stack.

Pops two words off the top of the stack, numerator first, and pushes their word-sized quotient onto the stack.

Pops two dwords off the top of the stack, numerator first, and pushes their dword-sized quotient onto the stack.

Pops two bytes off the top of the stack, numerator first, and pushes their byte-sized remainder onto the stack.

Pops two words off the top of the stack, numerator first, and pushes their word-sized remainder onto the stack.
**0x30 | 1 | mod dword**

Pops two dwords off the top of the stack, numerator first, and pushes their dword-sized remainder onto the stack.

**0x31 | 1 | and byte**

Pops two bytes off the top of the stack and pushes their byte-sized logical AND result onto the stack.

**0x32 | 1 | and word**

Pops two words off the top of the stack and pushes their word-sized logical AND result onto the stack.

**0x33 | 1 | and dword**

Pops two dwords off the top of the stack and pushes their dword-sized logical AND result onto the stack.

**0x34 | 1 | or byte**

Pops two bytes off the top of the stack and pushes their byte-sized logical OR result onto the stack.

**0x35 | 1 | or word**

Pops two words off the top of the stack and pushes their word-sized logical OR result onto the stack.

**0x36 | 1 | or dword**

Pops two dwords off the top of the stack and pushes their dword-sized logical OR result onto the stack.

**0x37 | 1 | xor byte**

Pops two bytes off the top of the stack and pushes their byte-sized logical XOR result onto the stack.

**0x38 | 1 | xor word**

Pops two words off the top of the stack and pushes their word-sized logical XOR result onto the stack.

**0x39 | 1 | xor dword**

Pops two dwords off the top of the stack and pushes their dword-sized logical XOR result onto the stack.

**0x3A | 1 | sgz byte**

Pops the top byte off the stack, and if this byte is greater than zero, the byte sized value 1 is pushed onto the stack, otherwise 0.

**0x3B | 1 | sgz word**

Pops the top word off the stack, and if this word is greater than zero, the byte sized value 1 is pushed onto the stack, otherwise 0.

**0x3C | 1 | sgz dword**

Pops the top dword off the stack, and if this dword is greater than zero, the byte sized value 1 is pushed onto the stack, otherwise 0.
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0x3D   | 1    | **sgez byte**  
|        |      | Pops the top byte off the stack, and if this byte is greater than or equal to zero, the byte sized value 1 is pushed onto the stack, otherwise 0. |
| 0x3E   | 1    | **sgez word**  
|        |      | Pops the top word off the stack, and if this word is greater than or equal to zero, the byte sized value 1 is pushed onto the stack, otherwise 0. |
| 0x3F   | 1    | **sgez dword**  
|        |      | Pops the top dword off the stack, and if this dword is greater than or equal to zero, the byte sized value 1 is pushed onto the stack, otherwise 0. |
| 0x40   | 1    | **sez byte**  
|        |      | Pops the top byte off the stack, and if this byte is equal to zero, the byte sized value 1 is pushed onto the stack, otherwise 0. |
| 0x41   | 1    | **sez word**  
|        |      | Pops the top word off the stack, and if this word is equal to zero, the byte sized value 1 is pushed onto the stack, otherwise 0. |
| 0x42   | 1    | **sez dword**  
|        |      | Pops the top dword off the stack, and if this dword is equal to zero, the byte sized value 1 is pushed onto the stack, otherwise 0. |
| 0x43   | 1    | **snez byte**  
|        |      | Pops the top byte off the stack, and if this byte is not equal to zero, the byte sized value 1 is pushed onto the stack, otherwise 0. |
| 0x44   | 1    | **snez word**  
|        |      | Pops the top word off the stack, and if this word is not equal to zero, the byte sized value 1 is pushed onto the stack, otherwise 0. |
| 0x45   | 1    | **snez dword**  
|        |      | Pops the top dword off the stack, and if this dword is not equal to zero, the byte sized value 1 is pushed onto the stack, otherwise 0. |
| 0x46   | 2    | **sll byte imm**  
|        |      | Pops the top byte off the stack, logically shifts it left by \(imm\) bits, and pushes the result onto the stack. |
| 0x47   | 2    | **sll word imm**  
|        |      | Pops the top word off the stack, logically shifts it left by \(imm\) bits, and pushes the result onto the stack. |
| 0x48   | 2    | **sll dword imm**  
|        |      | Pops the top dword off the stack, logically shifts it left by \(imm\) bits, and pushes it back. |
| 0x49   | 1    | **sllv byte**  
<p>|        |      | Pops the top two bytes off the stack, shifts the first logically left by the number of bits specified by the second byte, and pushes the result onto the stack. |</p>
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Size</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4A</td>
<td>1</td>
<td>sllv word</td>
<td>Pops the top word off the stack and then a byte off the stack, logically shifts the word left by the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>amount of bits specified by the byte, and pushes the result onto the stack.</td>
</tr>
<tr>
<td>0x4B</td>
<td>1</td>
<td>sllv dword</td>
<td>Pops the top dword off the stack and then a byte off the stack, logically shifts the dword left by the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>amount of bits specified by the byte, and pushes the result onto the stack.</td>
</tr>
<tr>
<td>0x4C</td>
<td>2</td>
<td>srl byte imm</td>
<td>Pops the top byte off the stack, logically shifts it right by (imm) bits, and pushes the result onto</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>the stack.</td>
</tr>
<tr>
<td>0x4D</td>
<td>2</td>
<td>srl word imm</td>
<td>Pops the top word off the stack, logically shifts it right by (imm) bits, and pushes the result onto</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>the stack.</td>
</tr>
<tr>
<td>0x4E</td>
<td>2</td>
<td>srl dword imm</td>
<td>Pops the top dword off the stack, logically shifts it right by (imm) bits, and pushes it back.</td>
</tr>
<tr>
<td>0x4F</td>
<td>1</td>
<td>srlv byte</td>
<td>Pops the top two bytes off the stack, shifts the first logically right by the number of bits specified</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>by the second byte, and pushes the result onto the stack.</td>
</tr>
<tr>
<td>0x50</td>
<td>1</td>
<td>srlv word</td>
<td>Pops the top word off the stack and then a byte off the stack, logically shifts the word right by the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>amount of bits specified by the byte, and pushes the result onto the stack.</td>
</tr>
<tr>
<td>0x51</td>
<td>1</td>
<td>srlv dword</td>
<td>Pops the top dword off the stack and then a byte off the stack, logically shifts the dword right by</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>the amount of bits specified by the byte, and pushes the result onto the stack.</td>
</tr>
<tr>
<td>0x52</td>
<td>2</td>
<td>sra byte imm</td>
<td>Pops the top byte off the stack, arithmetically shifts it right by (imm) bits, and pushes the result</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>onto the stack.</td>
</tr>
<tr>
<td>0x53</td>
<td>2</td>
<td>sra word imm</td>
<td>Pops the top word off the stack, arithmetically shifts it right by (imm) bits, and pushes the result</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>onto the stack.</td>
</tr>
<tr>
<td>0x54</td>
<td>2</td>
<td>sra dword imm</td>
<td>Pops the top dword off the stack, arithmetically shifts it right by (imm) bits, and pushes it back.</td>
</tr>
<tr>
<td>0x55</td>
<td>1</td>
<td>srav byte</td>
<td>Pops the top two bytes off the stack, shifts the first arithmetically right by the number of bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>specified by the second byte, and pushes the result onto the stack.</td>
</tr>
<tr>
<td>0x56</td>
<td>1</td>
<td>srav word</td>
<td>Pops the top word off the stack and then a byte off the stack, arithmetically shifts the word right</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>by the amount of bits specified by the byte, and pushes the result onto the stack.</td>
</tr>
<tr>
<td>0x57</td>
<td>1</td>
<td>srav dword</td>
<td>Pops the top dword off the stack and then a byte off the stack, arithmetically shifts the dword right</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>by the amount of bits specified by the byte, and pushes the result onto the stack.</td>
</tr>
</tbody>
</table>


