A Just-In-Time compiler for the BBC micro:bit

Thomas Denney
Supervisor: Alex Rogers
Word count: 9,989

Trinity 2018

Abstract

Cheap, low-powered programmable hardware, such as the BBC micro:bit, has lead to a renaissance of computer science education in schools; however many programming environments used by students either require the transfer of source code to be interpreted on the device, or the transfer of a much larger, compiled binary.

We demonstrate the use of a stack-based intermediate bytecode for programs that can be Just-in-Time (JIT) compiled on a BBC micro:bit to support a broad range of programming languages on a limited device, reduce the transfer requirement, and improve performance by avoiding interpretation.

The use of a JIT compiler on such a low-powered device demonstrates the possibility of software updates to sensors or Internet of Things (IoT) devices in environments with extremely limited network connectivity and power.

Acknowledgements

I am indebted to my supervisor, Professor Alex Rogers. His expert guidance ensured that the project was a success, particularly when it came to performing quantatitve performance measures of my JIT compiler. Our off-topic discussions during our meetings were also an absolute pleasure.

I'd also like to thank Steve Hodges and James Scott at Microsoft Research, without whom I wouldn't have been exposed to embedded programming or the micro:bit. Along with Professor Gavin Lowe, they have been amongst the best mentors that I could have hoped for.

Finally, I also owe a debt of gratitude to my friends and family, especially whilst I maintained a quirky — at best — sleep schedule working on this project.

Contents

Chapter 1

Introduction

The BBC micro:bit is a microcontroller designed to encourage schoolchildren to explore technology and programming [Mic16]. A number of educational tools support programming in block-based visual programming languages, interpreted languages, and natively compiled languages. Stack is an extensible bytecode for programs, developed at Oxford by Alex Rogers, for teaching stack machines and programming microcontrollers. It combines a programming model similar to traditional stack machines with the ability to add instructions that directly use device features such as LEDs or sensors. This project contributes a JIT compiler for Stack code executing on the micro:bit.

Stack is specified as a bytecode representation for programs that execute on the Stack Virtual Machine (VM), as described in Appendix A. The VM maintains an *operand stack* of integer values and a *call stack* of return addresses, and Stack instructions affect at least one of these stacks. For instance, the instruction `ADD` pops two values off the operand stack and pushes back their sum whilst the `CALL` instruction pops a value off the operand stack, pushes the address of the next instruction to the call stack, and jumps to the popped address.

1.1 Motivation

All existing programming tools for the micro:bit interpret source code or compile it Ahead-of-Time (AOT). To run a program on the micro:bit most of these tools require a binary to be 'flashed' via USB, followed by a device reboot, which often takes around 10 s. Interpreted code can be transferred to the device faster, as only the programmer's source code needs to be transferred, but at the expense of significantly worse performance. JIT compilation offers a compromise that requires the transfer of a very small amount of data compared to the full native binary, but the potential for comparable performance to native code, and far superior performance than interpreting code.

Reducing the transfer time for performant code benefits students as it makes it easier for them to build an intuitive mental model of how programming constructs work faster [Sen+16; Sen+17].

Demonstrating JIT compilation on a low-powered microcontroller suggests the technique could be applied to software on IoT devices. Long-range wireless protocols for IoT devices often permit payloads on the order of hundreds of bytes per day, which is much too small for native code, but large enough for one or more functions encoded in Stack bytecode [LoR18; Sig18].

1.2 Requirements

This project must implement an 'environment' that supports receiving Stack bytecode via a USB serial connection, JIT compiling that bytecode to native Arm code, and writing the compiled code to flash so that the micro:bit can reboot and later execute the same program.

The JIT compiled Arm code must exactly mimic the behaviour of the theoretical Stack VM, except in cases where physical limitations prevent perfect execution, or support for infrequently used Stack features would reduce performance in the general case. In the case of an error, the compiled code must return control to the surrounding environment with an appropriate error code.

Stack programs are often very small, and typical handwritten programs rarely have bytecode exceeding several hundred bytes. To support a typical range of Stack programs it must therefore be possible to transfer, compile, and execute a program of at least 500 B in size.

For JIT compilation to be worthwhile, the time taken to compile and execute Stack bytecode as native Arm bytecode must be comparable to the time taken to interpret the code, whilst the execution time must be significantly shorter. A reasonable goal, based on other JIT compilers, is for JIT compiled code to run 10 times faster than interpreted code [Pal17; PyP18; Ora18; Arn+04]. A series of experiments must demonstrate that this requirement is met. It will be necessary to develop a timing mechanism with external hardware; the micro:bit doesn't have a user-accessible clock of sufficient precision. The compiler will also need to support disabling certain features to allow for time comparisons between different optimisation modes.

1.3 Challenges and restrictions

Many JIT compilers 'trace' execution to determine which functions are executed most frequently and should therefore be compiled most optimally. Although tracing JIT are now standard for most major programming languages that use JIT compilation, a tracing JIT will *not* be demonstrated on the micro:bit due to the limited performance of the device and relatively small size of Stack programs; tracing JIT are generally most useful in large programs [Arn+04; Bol+09; Ora18].

A compiled Stack program will require no more memory than an interpreted program, aside from the additional space to store the compiled bytecode. However, the compiler will require additional memory to write the compiled bytecode to and for its own internal data structures. Although the micro:bit has 16 kB of RAM, only around 8 kB is available to the programmer.

The low performance of the micro:bit and the need to compile quickly restricts the variety and complexity of compiler techniques that can be used. Many traditional compiler techniques depend on an abstract syntax tree representation of the program code for tasks such as static analysis, type checking, and instruction selection [App98]. However, the construction of these trees is not viable with so little time and memory available, so techniques developed will rely on an implicit tree or other elimination methods. Many compilers are implemented in high-level languages, which often require large amounts of memory to support features such as garbage collection. Limited performance and the availability of compilers therefore makes low-level languages such as C, C++, or Rust a necessity [Lan16; Sap16].[1]

1.4 Contributions

This project introduces a JIT compiler for the micro:bit, which is also the first JIT compiler for the Cortex-M0 processor. The JIT compiler can be embedded within existing micro:bit programs, or alternatively used in conjunction with an IDE developed for Stack to support rapid redeployment of programs.

The new redeployment process allows a program to be deployed immediately to the device and JIT compiled so that it can begin execution in under 100 ms, compared to 10 s for a full redeployment of a native binary, which is typical of other tools.[2] Additionally, JIT compiled code can execute 10-50 times faster than the interpreter.

Considerable effort was directed at the correctness and performance of the JIT compiler. Over 400 tests were used to ensure that the compiler produced correct code, and a quantitative testing approach was employed to measure the exact execution time of compiled code.

Finally, we consider future extensions of this project, including the possibility of using JIT compilation on microcontrollers that use low-powered long-range wireless networks to support software updates on IoT sensors.

[1]Haskell and OCaml have been demonstrated on more powerful microcontrollers, but there are currently no major implementations for either language that support the micro:bit [Gre18; Vau12].

[2]It takes 9 s to deploy a new binary to the micro:bit, and all other tools take at least 1 s to compile a new native binary. 10 s is therefore a generous lower bound, as students also have to copy the compiled file to the device.

1.5 Outline of the report

- Chapter 2 summarises the BBC micro:bit, the Arm architecture and Cortex-M0 processor, calling conventions, JIT compilation, and related work
- Chapter 3 discusses the implementation of the project's software, including the software that receives code via serial and the compiler and its low-memory compilation techniques
- Chapter 4 discusses the correctness testing of the compiler and the performance testing, which compared the interpreter with the JIT compiled code,
- Chapter 5 discusses possible future extensions of this project
- The appendices include a description of the Stack VM, Arm Thumb instructions, JIT compiler options, and some tests

The source code for this project is available at `http://github.com/thomasdenney/microjit`.

Chapter 2

Background

2.1 The BBC micro:bit

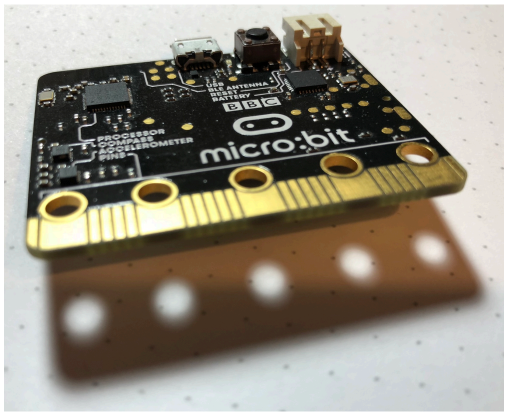

Figure 2.1: The BBC micro:bit

The micro:bit was developed by the BBC and partner organisations to “inspire digital creativity and develop a new generation of tech pioneers,” and several programming environments were developed to support it [BBC15]. A C++ SDK, which wraps the lower-level SDKs for interacting with device features such as the Bluetooth radio, was used for this project. The C++ SDK is wrapped by high-level APIs in JavaScript and Python, which in turn support block-based programming environments targeted at children [Lan16; Mic18].

The micro:bit emulates a FAT filesystem when plugged into a computer. In most programming environments the programmer must transfer a compiled binary file to the micro:bit, which then writes the program to its internal flash and restarts before running the program. There have been proposals to transfer only the code that changed to the micro:bit, but these were largely discarded as too complex [FMM16]. Conversely, the MicroPython implementation supports a Python REPL over serial, but it is restricted to interpreting a limited subset of Python [GB18].

The micro:bit has drawn comparison with the Raspberry Pi, another British low-powered Arm-based computer aimed at encouraging children to pursue CS [Ras18]. The micro:bit was designed to be simpler and lower powered — it doesn't require the installation of a Linux distribution! The micro:bit can interact with a Raspberry Pi via serial, or General Purpose Input Output (GPIO) pins [Haw16].

There are two processors on the micro:bit [Mic17a]. The first is a Freescale-developed Cortex-M0+ core that is used exclusively for USB communication. A second Cortex-M0 core, Nordic Semiconductor's nRF51, is used for code execution and interfaces with all other hardware. The main Cortex-M0 core is clocked at 16 MHz whilst the secondary Cortex-M0+ core is clocked at 48 MHz [Fre14; Nor14].

The micro:bit has 256 kB of flash storage and 16 kB of RAM [Mic17a]. Both are addressed using 32-bit virtual addresses. It is possible to overwrite around 50 kB of empty flash storage at runtime.[1]

The primary communication interface used by this project was the micro:bit's serial link, which permits sending and receiving binary data over a USB link with a PC. High-level APIs on the micro:bit allow this interface to be treated like `stdout`/`stdin`. On a PC, software such as `tty` can be used for communicating with the micro:bit [Lan16]. The micro:bit also has a set of digital GPIO pins, which were used in this performance testing, as discussed in Section 4.2.

2.2 Arm architecture

Arm Holdings designs the Arm architecture and Arm core designs, licensing both to chip manufacturers and industry partners. Arm's designs follow the Reduced Instruction Set Computer (RISC) architecture. The designs are split into the A-series for general purpose computing and mobile phones, the R-series for real time applications, and the M-series for lower-powered embedded devices [Arm18a]. Many of the M-series cores support Wi-Fi, and are widely deployed in IoT products, but the Cortex-M0 and -M0+ are the smallest, cheapest, and most energy efficient of all the designs in the series [Arm18c; Arm18d].

Recent A-series cores implement the 32-bit ARMv7 or 64-bit ARMv8 architectures; both of these encode instructions in 32 bits. In an embedded environment this is often untenable due to restricted flash capacities, an increased instruction decoding cost, and a larger bus size requirement. Competitive chips have 8-bit or 16-bit instructions [Arm05a; Atm16]. The Cortex-M0 implements the Arm Thumb architecture, which encodes most instructions in 16 bits.

Although it has a distinct instruction set encoding, the Thumb architecture shares a lot of similarities with the full 32-bit Arm architecture. Both make 16 registers accessible to the programmer, but in the Thumb architecture only the lower 8 registers are general-purpose (this reduces the number of bits needed for register names). The remaining upper eight registers can be accessed by a limited subset of instructions, although three have special purposes [Arm05a]:

[1]This value was determined by a runtime test in the compiler.

- `r15` is the program counter
- `r14` is the link register
- `r13` is the stack pointer

2.2.1 Calling convention

The procedure call standard for the Arm architecture passes the first four arguments that fit in 32-bits in the first four registers, and pushes the remainder of the arguments to the stack. The link register, `r14`, contains the return address and by branching back to this address a function can return. The remainder of the registers are saved by the callee, and the function result is usually returned in `r0` [Arm15].

```
@ The link register only needs to be pushed if this function makes a
@ call to another function, which would cause its value to be
@ overwritten
push lr, r4, r5, r6, r7

@ Function code goes here

@ Pops the value of the LR back to the PC. Alternatively the instruction
@ blx lr can be used to return
pop pc, r4, r5, r6, r7
```

Listing 2.1: A skeleton for a function in Arm Assembly

2.2.2 Arm Cortex-M0

The nRF51 core used on the micro:bit is an Arm Cortex-M0 core clocked at 16 MHz, so each cycle executes in 6.25×10^{-8} s.[2] The majority of instructions execute in a single cycle, but all instructions requiring memory accesses (e.g. loads or stores) take at least two cycles. This contrasts significantly with memory accesses on modern desktop processors, which can take 4-300 cycles, depending on whether the data is cached [Int17]. Branches take 1-3 cycles (depending on whether the branch is taken).[3]

The relatively low latency of memory operations aids compiler development, as the cost of *spilling* registers (saving register values to memory so that the register can be used for storing another value) is cheap. Therefore relatively straightforward register allocation algorithms can be used.

[2]Tests suggest that the clock rate is just below 16 MHz

[3]A full list of cycle counts can be found in Appendix B.

The Cortex-M0 doesn't feature many optimisations typical of modern processors, such as out-of-order execution and branch prediction, so the time a sequence of instructions takes to execute is deterministic. However, the processor is pipelined such that instruction fetching, decoding, and execution can occur in parallel [Arm10].

The majority of Thumb instructions are encoded in 16 bits, with the exception of a few branch instructions encoded in 32 bits. Thumb-2 is a later version of the instruction set that adds more 32 bit encoded instructions, including more instructions that can be conditionally executed. The Cortex-M0 implements very few Thumb-2 instructions, and none of them are used in my JIT [Arm10].

2.3 Stack

'Stack' is a VM specified by Alex Rogers and used at outreach events by the Department of Computer Science, Oxford [Rog17a]. The VM is similar to traditional stack machines but its instruction set is extensible; it supports adding optional instructions without needing to change existing runtimes. The VM maintains an *operand stack* and a *call stack* with high-level instructions to manipulate both.

Most Stack instructions are encoded in a single byte, as documented in Appendix A. The only exceptions to this are the `PUSH` instructions, which can take a signed 1- or 2-byte operand, and optional instructions, which encode their effect on the stack in a second byte. Like the Thumb instruction set, the Stack instruction set aims for very high code density, but sacrifices low-level efficiency for high-level stack operations. See Section 4.3 for a comparison of their code density.

Stack was intentionally developed to support the hardware features available on microcontrollers such as the BBC micro:bit, and therefore supports several instructions that access LEDs and the accelerometer. Additionally, Stack supports using an Adafruit Neopixel, a programmable array of RGB LEDs, with the micro:bit [Ada18].

Prior to this project there were two major implementations of the Stack VM, both of which interpreted Stack bytecode rather than compiling it to native machine code [Rog17b]. The first was implemented as part of web app that additionally featured an assembler and linker, allowing the programmer to compile a text-based assembly language to Stack bytecode. The second executed on the micro:bit, but required that the Stack code was included as part of the binary transferred to the micro:bit for execution. Importantly, this meant that a user that wanted to execute their code on the micro:bit had to transfer a 512 kB file to their micro:bit, wait for it to fully flash the device, and reboot it in order to see their code execute.[4]

[4]The file encodes the new contents of the micro:bit's 256 kB flash storage in ASCII hexadecimal.

2.3.1 Comparison with similar virtual machines

Stack's assembly language shares a great deal of similarities with Forth, a standardised concatenative stack-based programming language originally developed in the 1970s for use on computers with limited memory [RCM93]. It is still used in some microcontroller software, and later inspired similar concatenative languages such as PostScript, which remains widely used in printers [Ert99].

The semantics of Forth are similar to Stack, in that it maintains separate call stacks and operand stacks, but its treatment of functions differs.[5] A function call in Stack corresponds to pushing an address to the operand stack and issuing the `CALL` instruction, which will then push the return address to the call stack, pop the function address from the operand stack, and jump to that address. Forth instead permits the programmer to define a function as a new 'word' in the language that has an explicit effect on the stack (e.g. a function that squares a number pops one integer and pushes its square back to the stack), and then allows that 'word' to be used like any other language instruction. Additionally, Forth provides explicit control structures (e.g. `IF THEN ELSE`) whilst Stack only provides jumps. In general, Stack is more low-level than Forth and it would be possible to create a compiler from a restricted subset Forth to Stack — Forth features such as self-modifying code would have to be disabled.

Many other VMs are based on stack machines, including .Net's Common Language Runtime and the Java Virtual Machine [Mic17b; Lin+15]. They each specify a bytecode format and instruction set with semantics designed for high-level languages with features such as garbage collection. Both VMs allocate stack frames for each function call, which contains an array of local variables and a separate operand stack.[6] Stack doesn't separate the local variables and the operand stack, nor does it explicitly separate the stack frames of different functions. Stack's lack of such metadata and simpler instruction set allows for smaller program code, although at the cost of making verification (e.g. type checking, bounds checking) significantly more complex as metadata must be derived from program code.

2.4 Compilation techniques

Typical compilers are structured into a front end, which parses program source code and handles high-level tasks such as type checking, and a back end which performs static analysis, code generation, and linking [App98]. The front end typically outputs an intermediate representation, which may be the bytecode for a VM or an abstract syntax tree [LLV18a].

In compiler nomenclature, a *basic block* is a sequence of instructions that are always entered at the same point, executed in the same order, and are terminated by a branch instruction (which may be an implicit branch to the subsequent basic block). Only the heads of basic block are permissible

[5] Forth calls the call stack and operand stack the 'return stack' and 'data stack' respectively.

[6] Note that the approach for allocating stack frames is not specified by the specifications; some Java implementations may heap-allocate stack frames, for example.

jump or call destinations [App98].

Most compilers generate assembly for their target architecture in a text format and rely on an external assembler and linker to convert this to the native machine code representation and resolve jump and call destination addresses [GCC18; LLV18b]. Relying on external tools in this project is not possible as most compiler toolchains are too large to execute on the micro:bit itself.

2.4.1 JIT compilation

JIT compilation compiles an intermediate representation of a program to native machine code at the moment the code needs to be executed. Thanks to the growth in popularity of languages such as Java and JavaScript since the mid-1990s, JIT compiled programs are executed on billions of computers [Goo18]. Typical benefits include improved code density and portability, whilst significantly improving on the performance of interpreting code instead, as described in Section 1.2. Many modern JITs trace program execution and use more aggressive compiler optimisations where functions — and even basic blocks — are executed more frequently [Piz16].

2.5 Related work

A number of AOT and JIT compilers target the Arm architecture, and in many cases these compilers generate Thumb instructions rather than full Arm instructions [LLV18a; Mon18]. Existing JIT compilers only target the A- series of processor cores, which are commonly used in smartphones and need to efficiently execute JIT compiled languages such as Java and JavaScript. There have been no substantial efforts to implement JIT compilers on Arm R- or M- cores as these are substantially lower-powered and it is has only recently become common to program them in higher-level languages.

Several other programming environments exist for the micro:bit. Microsoft's MakeCode supports JavaScript and a visual "block-based" mode. It directly compiles the JavaScript to a Thumb binary which must then be transferred to the device [Mic18]. MicroPython implements a limited subset of Python 3 and interprets code directly on the device [GB18].

Chapter 3

The JIT compiler

I implemented a JIT compiler for Stack bytecode in C++. The compiler directly generates binary-encoded Arm Thumb instructions, a necessity discussed in Section 2.4. An interpreter, loosely based on prior work by Rogers, was developed alongside the compiler and was used to verify the behaviour of the compiled code. Like the interpreter, the compiler takes a pointer to an array of Stack bytecode, but will then return a pointer to generated Thumb bytecode on success rather than immediately interpreting the code.

The compiler and VM are separated into a traditional, modular structure [App98]:

- **Stack instruction decoding:** Stack instructions are represented in the compiler through an enumeration type. Additionally, a special 'iterator' allows for forward and reverse iteration through Stack bytecode. Importantly, this class abstracts over variable-length and push instructions.

- **Device Abstraction Layer (DAL):** To simplify the generated code and to avoid dealing with C++ calling conventions and the micro:bit SDK, a set of simpler functions were designed to abstract over many of the optional Stack instructions. Some arithmetic operations (`DIV` and `MOD`) were also implemented in this manner if there were no hardware implementations, and some stack operations (`NDUP`, `NROT`, `NTUCK`) fall-back to C implementations.[1]

- **Static analysis:** Before compilation proceeds each instruction reachable from the first byte in the code/data array is annotated with metadata, including whether or not it is the start of a basic block, the start of a function, or whether its function uses recursion. The final result of static analysis is a list of all reachable functions and their basic blocks. As described in Section 3.8, the static analyser can be run at a later point from a different offset (to discover previously unreachable code) for lazy compilation.

[1]See Section 3.6 for further discussion.

- **Compiler:** Once static analysis is complete the compiler generates boilerplate code for handling entry and exit to JIT-compiled code and bounds checks before generating Arm bytecode based directly on the structure discovered in the static analysis phase.

- **Thumb bytecode encoder:** A set of around 70 functions corresponding to each of the instructions supported on the Cortex-M0 encoded the operands of each instruction. Most of these functions perform basic bit shifts, and can therefore be inlined by the compiler.

- **Linker:** Used concurrently with the compiler. The compiled location of each basic block is recorded by the compiler. Whenever a branch or call is encountered a series of `nop` instructions are emitted. Afterwards, the linker then replaces the `nop` instructions with branches and calls to the correct destination.

Once linking is complete, the entire Arm bytecode for the compiled Stack code resides in a buffer in memory. Before the code can be executed it is necessary to issue the `DSB` and `ISB` instructions, which ensure that all memory writes complete and the instruction pipeline is flushed [Arm05a; Arm12; Bra13]. The Cortex-M0 doesn't have a cache, but it does have a 3-stage pipeline, so it's possible that the processor attempts to read the JIT-compiled code before it's written back to memory. These instructions ensure all JIT-compiled code is fully written back to memory before they are read back for execution. Section 3.4 describes how the bytecode in the buffer behaves exactly like a function that adheres to Arm calling conventions, which means that the address of the buffer can be cast to a function pointer, and then directly called.

Additionally, separate modules were developed to handle the transfer of Stack bytecode over serial and writing the compiled result to flash. Separate code was developed to decode Arm instructions and verify the behaviour of the compiler with respect to the interpreter across over 400 tests, described in chapter 4.

An important consideration, described in Section 1.3, is that individual features and optimisations of the compiler can be enabled and disabled as needed. All configuration is done at the compile-time of the compiler itself, so that the performance of different code generation techniques can be compared. Compiler options control the approach for compiling conditional branches, register allocation, constant loads, bounds checks, and tail-call optimisation. They are fully described in Appendix C.

3.1 Execution flow

In order to execute code with the interpreter, a client directly calls the interpreter with a pointer to the stack code to be compiled. It then iterates through the code using the Stack bytecode decoder and executes instructions as appropriate. It possibly also calls into the DAL for the execution of certain instructions:

$$\begin{tikzcd}
 & & \text{Entry point} \arrow[d] & \\
\text{Decoder} & \text{Iterator} \arrow[l] & \text{Interpreter} \arrow[l] \arrow[r] & \text{DAL}
\end{tikzcd}$$

Figure 3.1: Interpreter flow

On the other hand, when the compiler is called it will call the static analyser, which will in turn call the code iterator and decoder. The compiler itself then iterates through each function and basic block, and encodes Thumb instructions before linking all the jumps. This then produces bytecode that can be called:

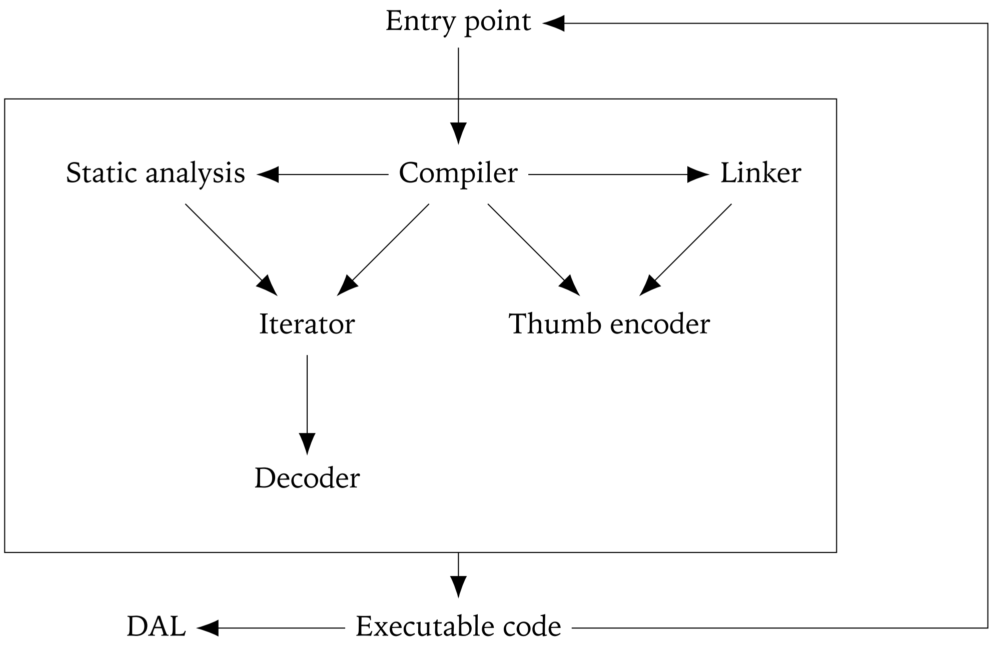

Figure 3.2: Compilation and execution flow

When the project is used for programs that are sent via the USB serial link, the execution path for the device is (lazy compilation means that execution can also enter the compiler, as in Figure 3.2):

$$\begin{tikzcd}
\text{Start up} \arrow[r] & \text{Execute code (if any) from flash} \arrow[r] & \text{Wait for new code} \arrow[d] \\
& \text{Write new code to flash} \arrow[u] & \text{Compile new code} \arrow[l]
\end{tikzcd}$$

Figure 3.3: Serial transfer flow

3.2 Compiling a subset of Stack

The Stack specification is deliberately straightforward with very few instructions, however there are a few potential ambiguities and edge cases that would require significantly more memory in order to compile (due to the need for extra metadata about the bytecode) or would compile to very inefficient code.

The most important restriction is limiting conditional and unconditional jumps to those with 'constant' destination addresses, i.e. the `JMP` and `CJMP` instructions that are immediately preceded by a $\texttt{PUSH}_8$ or $\texttt{PUSH}_{16}$ instruction.[2] This restricts jump destinations to addresses in the range $0 \leq a < 2^{15}$ as both `PUSH` instructions push signed values, and negative addresses are not permitted. In the case of the micro:bit this doesn't particularly matter because all compiled code must fit within RAM, which has a capacity of 16 kB — each Thumb instruction takes two bytes so it would never be possible for us to compile more than 2^{13} Thumb instructions.[3] The main reason for this is that it significantly reduces the complexity of the linker, and avoids the potential for jumping into the middle of basic block, as the compiler assumes that a basic block is only entered at its beginning and only exited at its end.

Section 3.8 shows that this restriction need not apply to function calls.

Finally, the basic blocks of an individual function must be contiguous in the original Stack code. This assumption is entirely reasonable as it is unlikely that a compiler or human would choose to overlap basic blocks of different functions.

Other than these restrictions, the compiler generates code that exactly mimics the behaviour of the definitional interpreter.

[2]This restriction also implies that a conditional/unconditional jump instruction cannot itself be a jump destination

[3]In reality the maximum program size is significantly smaller than this as the micro:bit runtime consumes several kilobytes of RAM and the compiler's own data structures also prevent larger sizes being available.

3.2.1 Static analysis rules

The above properties are extended to a general rule that permits each byte in the Stack program to be only interpreted in a single way. Consider the following Stack code, along with its binary encoding:

```
Offset  Instruction Encoding
0       Push8 24    0x18 0x18
2       Push8 32    0x18 0x20
5       Mul         0x02
6       Halt        0x20
```

If a piece of Stack code jumped to offset 0 then the stack would be filled with $24 \times 32 = 768$ before halting. If another piece of code jumped to offset 1 it would then interpret byte 1 as the start of a push instruction, and its push value as `0x18`, and byte 4 as a halt instruction. This would leave 24 on the stack, which is not desired. Variable length instructions therefore allow each byte to have many interpretations depending on the start offset (consider a longer sequence of PUSH_8 and PUSH_{16} instructions), and it would add a great deal of additional unnecessary complexity to the static analyser and compiler to determine which interpretation is the correct one to jump to.

Let

- n denote the offset of an instruction in an array of instructions of capacity K;
- $\mathsf{next}(n)$ be a function for the offset of the next instruction;
- $\mathsf{prev}(n)$ be a function for the offset of the previous instruction;
- $\mathsf{pushValue}(n)$ be a function for the value pushed by a push instruction at n;
- $\mathsf{INS}(n)$ be a predicate for each Stack instruction INS. Let $\mathsf{Code}(n)$ hold if and only if $\mathsf{INS}(n)$ holds for some INS.
- $\mathsf{Unexecutable}(n)$ indicates that the byte at offset n should never be interpreted as an executable instruction; and
- $\mathsf{BasicBlockStart}(n)$ indicates that n is at the start of a basic block whilst $\mathsf{FunctionStart}(n)$ indicates that n is that the start of a function.

The **single interpretation rule** then requires the following property of the code:

$$\mathsf{Code}(n) \Leftrightarrow \neg\mathsf{Unexecutable}(n)$$

Additionally, the static analyser also requires the following:

- $\mathsf{Code}(0) \wedge \mathsf{BasicBlockStart}(0)$, i.e. the first byte is the start of some basic block that can be entered
- $\mathtt{INS}(n) \Rightarrow \mathsf{Code}(n)$ for all instructions INS
- $\mathsf{BasicBlockStart}(n) \Rightarrow \mathsf{Code}(n)$
- $\mathsf{FunctionStart}(n) \Rightarrow \mathsf{BasicBlockStart}(n)$
- $\mathsf{PUSH}_8(n) \Rightarrow \mathsf{Unexecutable}(n+1) \wedge n+1 < K$
- $\mathsf{PUSH}_{16}(n) \Rightarrow \mathsf{Unexecutable}(n+1) \wedge \mathsf{Unexecutable}(n+2) \wedge n+2 < K$
- $\mathtt{OPT}(n) \Rightarrow \mathsf{Unexecutable}(n+1) \wedge n+1 < K$ for all optional instructions OPT
- $\mathsf{Code}(n) \wedge \neg(\mathsf{RET}(n) \vee \mathtt{JMP}(n) \vee \mathtt{HALT}(n)) \Rightarrow \mathsf{Code}(\mathsf{next}(n))$, i.e. unless this function halts control flow the next instruction should be also be an executable instruction
- Unconditional and conditional jumps must be preceded by a push instruction, that the push instruction pushes a value within the bounds of the instruction array, and that the corresponding destination instruction is the start of a basic block:

$$\begin{aligned}\mathtt{JMP}(n) \vee \mathtt{CJMP}(n) \quad \Rightarrow \quad & \mathsf{prev}(n) \geq 0 \wedge \mathsf{PUSH}(\mathsf{prev}(n)) \wedge \\ & 0 \leq \mathsf{pushValue}(\mathsf{prev}(n)) < K \wedge \\ & \mathsf{BasicBlockStart}(\mathsf{pushValue}(\mathsf{prev}(n)))\end{aligned}$$

- If a function call is preceded by a push then the corresponding instruction must be the start of a function:

$$\begin{aligned}\mathtt{CALL}(n) \wedge \mathsf{prev}(n) \geq 0 \wedge \mathsf{PUSH}(\mathsf{prev}(n)) \quad \Rightarrow \quad & 0 \leq \mathsf{pushValue}(\mathsf{prev}(n)) < K \wedge \\ & \mathsf{FunctionStart}(\mathsf{pushValue}(\mathsf{prev}(n)))\end{aligned}$$

Static analysis checks verify that these rules are never broken, and these checks are performed whenever lazy compilation is performed because it is possible that a lazily compiled function may attempt to violate the rules.

The static analyser allocates a separate array to store whether the predicate is true for each offset in the program being compiled. As only a single bit is required per predicate it is possible to store all the predicate values for a particular instruction within a single byte. Therefore for a program of size K the static analyser will only require an additional K bytes.

3.3 Virtual machine state

In order to model the behaviour of the Stack VM, a data structure, henceforth $\texttt{Stack}_{\texttt{Env}}$, is used to store the following pointers and values:

- $\texttt{Stack}_{\texttt{SBase}}$: base pointer for the operand stack — used in stack overflow checks
- $\texttt{Stack}_{\texttt{SP}}$: stack pointer for the operand stack
- $\texttt{Stack}_{\texttt{SEnd}}$: end pointer for the operand stack — used in stack underflow checks
- `Status`: a value that can be checked on termination indicating whether the VM terminated successfully or with some error
- $\texttt{Error}_{\texttt{PC}}$: the value of the program counter when an error occurred; used for debugging
- `Compiler`: a pointer to the data structure that was used to compile this code. This is required to support lazy compilation; see Section 3.8
- $\texttt{Escape}_{\texttt{SP}}$: the value of the Arm stack pointer when execution of the VM started. This is required for escaping out of a Stack program with the `HALT` instruction regardless of how deep the stack is. See Section 3.5.

3.4 Calling convention for Stack programs

Stack programs are compiled to Arm functions that can be called by code written in C/C++. Control flow returns to the calling function when the Stack program halts. A small amount of boilerplate code is generated to handle entry into the Stack code and returning to the main executable afterwards.

In order to support safe execution of Stack code, executing code must be able to access data about its environment, such as the status of the Stack VM or the bounds of the stack so it can determine if stack underflow or overflow has occurred. Therefore a pointer to the object storing the Stack environment is always stored in register `r0`.

The most significant difference between the Stack VM and Arm architecture is the number of stacks. Whilst the Stack VM maintains two separate stacks for operands (along with any other function data) and return addresses, Arm only has a single general-purpose stack, albeit with a number of instructions to simplify pushing/popping the return address.

Unlike other programming languages and environments, any piece of Stack code can access the full operand stack, so there is no distinction between global and local variables, parameters, and operands on the stack. As such, interleaving the operand stack with the call stack is impractical because it would mean that locating values on the stack beyond the return address would have to be dynamically resolved at runtime, rather than computing them from fixed offsets from the stack pointer.

The Stack runtime therefore maintains two separate stacks explicitly, just as the earlier interpreter did. In order to make efficient use of Arm instructions, the existing stack is used for the Stack call stack, whilst a separate buffer is used for the Stack operand stack. A pointer, storing the base address of the operand stack, is always stored in register `r1`. The register `r3` is additionally used as a temporary register.

Finally, in order to reduce the number of memory accesses required, the value of the top of stack is always stored in register `r2`. Whilst executing compiled Stack it is often necessary to execute a function that is part of the runtime, so by storing these three values in the three lowest registers they will form the first three arguments of any runtime function called, as per Section 2.2.1.

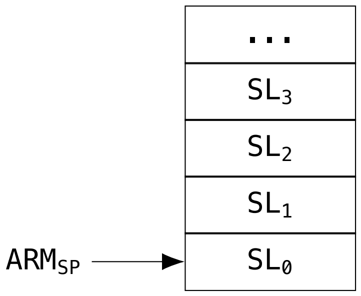

Figure 3.4: ARM stack state

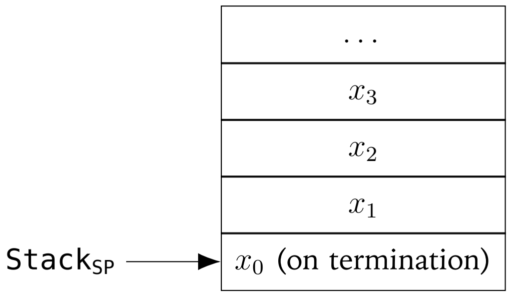

Figure 3.5: Operand stack state

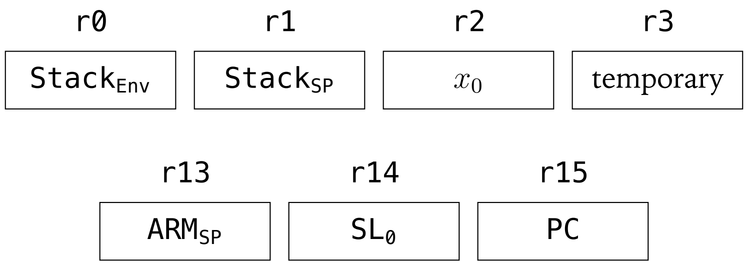

Figure 3.6: Register state

SL_i is the static link (return address) for the ith function on the stack.

The manner in which the return address is written to the call stack also differs between Stack and Arm. In Stack the return address is written on the `CALL` instruction (i.e. the caller writes it), whilst in Arm the called function *can* write it to the stack (and should if it then makes calls to other functions). Therefore an important feature of the static analyser is to record which Stack instructions are the destinations of `CALL` instructions so that it can be determined when, if at all, to write the return address to the stack.

3.5 Boilerplate code

When Stack code reaches the `HALT` instruction it is necessary to return control to the VM environment, in the same way that the existing interpreter does. Therefore when compiled code is entered the link register (which holds the return address to the calling C/C++ code) is pushed to the Arm stack along with all callee-saved registers, and the value of the Arm stack pointer is saved to $\texttt{Escape}_{\texttt{SP}}$, which is a field of the $\texttt{Stack}_{\texttt{Env}}$ data structure described in Section 3.3. Then, when the halt instruction is reached it is only necessary to restore the stack pointer to the saved value, and then return from the generated code by popping the original value of the link register to the program counter along with the callee-saved registers.[4]

Stack code can either halt by reaching the last instruction in its buffer or the `HALT` instruction. Therefore each function is followed by an unconditional branch to the halt code.

3.6 Register allocation

The compiler uses three different approaches for register allocation. These approaches can be switched between at compile time, as described in Appendix C, and their performance is compared in Section 4.2. The goal of a good register allocator is to ensure that frequently accessed values remain in registers, so that they don't need to be repeatedly accessed from memory, and secondly to ensure that there are sufficient free registers such that values don't have to be 'spilled' to memory [Aho+06]. Register allocation is generally split into two problems: *local* allocation within a basic block and *global* or *intra-block* allocation between basic blocks [Koo94]. Only local allocation algorithms were developed in this project, although it would be possible to extend the algorithm for algorithm for intra-block allocation.

Stack doesn't make a distinction between parameters, local variables, and temporary values, but most traditional register allocation algorithms depend on this information to generate more efficient code. More complex register allocation algorithms use graph colouring techniques to reduce the number of spills; these approaches also weren't implemented.

3.6.1 The straightforward approach

As described in Section 3.4, the VM state, stack pointer, and top of stack value are always stored in registers `r0`, `r1`, and `r2` respectively. The top of stack value and stack pointer only need to be written to memory when returning back to the calling C/C++ code. The first approach compiles Stack instructions so that their behaviour on memory mimics what the interpreter would do:

[4]In other environments, C++ exceptions are commonly used to solve the same problem; they are disabled in the ARM mbed compiler.

- When an instruction acts on the top of stack value the instruction(s) that perform the same operation are emitted without the need to affect memory or the stack pointer. For instance, the Stack instruction `INC` is compiled as `add r2, r2, #1` only

- For instructions that are binary operations, the stack pointer is incremented and the second stack value is loaded from memory into the temporary register `r3`. The operation is then computed into `r2`, the top of stack register. This means that addition, subtraction, and multiplication can be performed in 3 instructions or 5 cycles (see Appendix B)

- When a value needs to be pushed onto the stack the current top of stack value is written to memory, the stack pointer value is decremented, and the value is written to `r2`

As shown in Section 4.2, even this basic approach offers significantly better performance than the interpreter. However, it has fairly significant disadvantages:

- Conditional jumps are inefficient. The sequence of Stack instructions `< 5 CJMP`, which jumps to location 5 if the second value on the stack is less than the first, requires several conditional jumps because the value of `0` or `1` must be written to the operand stack by the compilation of the `<` operation, and this value must then be compared to 0 for the `CJMP` operation. If the two top of stack values were both in registers beforehand this operation could be reduced to a single comparison and branch.

- Most operations will result in a memory read or write, which means that the lower bound for most Stack instructions is 4 cycles when compiled to Arm instructions.[5]

3.6.2 Using more registers

An obvious extension of the ideas presented in Sections 3.4 and 3.6.1 is to use the remaining four general-purpose registers for storing more values from the operand stack.[6] The second implementation of register allocation supports this idea by permitting up to five values to be stored in registers instead of in memory. It also only updates the stack pointer when a value is written back to memory, which significantly reduces the number of memory writes.

When five operand stack values are stored in registers it may be necessary to spill one of the register values so that the register can then be reused. It is preferable to spill the value that is least likely to be used next. The compiler employees a simple heuristic: most Stack instructions only manipulate the top 2-3 values on the operand stack, so it always spills the deepest value still held in a register.

In the worst case, performance is exactly the same as the straightforward approach. However, in the best case it eliminates at least one load or store per Stack instruction, which can significantly reduce the total number of cycles required. Furthermore, this approach can reduce sequences of instructions such as `< 5 CJMP` to a single comparison and conditional branch.

[5]A load or store is 3 cycles, and all other operations take at least 1 cycle

[6]Recall from Section 2.2 that only the lower eight registers are accessible by all instructions.

3.6.3 Copy-on-write allocation for register values

Typical Stack code features a significant number of instructions that do nothing but manipulate values on the stack so that the operands of an arithmetic instruction are the top two values on the stack. In both of the previous approaches, the `SWAP` instruction would require at least three move instructions, for example.[7] If we know which registers hold which stack offsets, the compiler's internal data structures can be updated instead, which eliminates the need for register-to-register moves. `TUCK` and `ROT` are eliminated in a similar manner.

This approach can be further extended to eliminate the `DUP` instruction. Instead of maintaining only a single map of which registers are used for which stack offsets, we maintain a *read map* and a *write map*, which are maps from stack offsets to registers. Whenever a value is written back to the stack we simply update the read map so that it corresponds to the same register as the write map.

For instance, when a `PUSH` instruction is executed the read and write registers become equal:

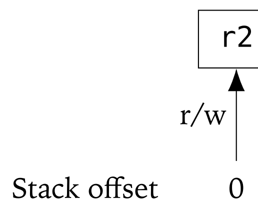

Figure 3.7: `PUSH` instructions cause the top of stack to have the same read/write register

However, when a `DUP` instruction is executed the read and write registers can be different:

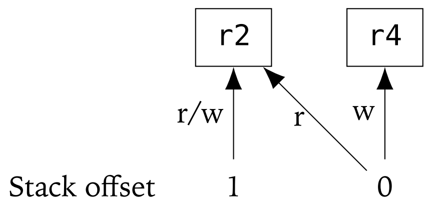

Figure 3.8: `DUP` instructions cause the top of stack to have different read/write registers

[7]The value a has to be moved to a temporary register, then b's value is moved to a's register, and finally a's value is moved from the temporary register to b's original register.

Spilling becomes a little more complex with this technique, because the read register for the deepest stack value may have multiple 'readers'. If this is case, the value is written back to the stack, and some higher stack offset that is still reading from the register has its write register updated to use the write register of the value just written back to memory, thereby freeing a register:

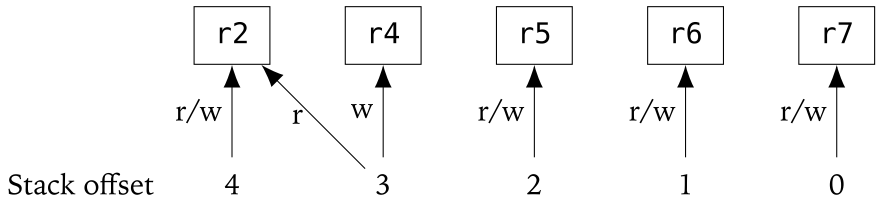

Figure 3.9: An example of the state before spilling

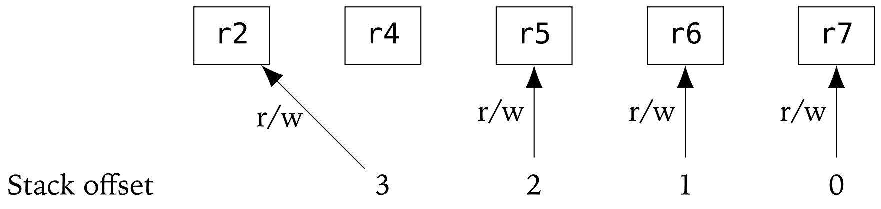

Figure 3.10: The resulting state that leaves register r_4 free

This approach immediately reduces the number of instructions emitted in cases where instructions do a lot of manipulation of the stack (because it effectively converts them to no-ops), but it was also extended to improve instruction selection for arithmetic operations. In many cases at least one operand of an arithmetic operation may be known in advance, so an additional map is used to store *known register values*. When a register whose value is known is used in an arithmetic operation, we may instead choose to use an instruction that can carry the register value as an intermediate operand.[8] If this is not possible, the value can instead be written to a register and an instruction that operates on two registers can be emitted instead. This approach also eliminates eliminate constant expressions, as if both operands to an operation are known at compile time then there is no need to compute them at runtime.

None of the approaches described support intra-block allocation, as register values are always committed to memory at the end of a basic block. A better approach would re-order (if necessary) register values so that they are consistent on basic block entry and re-entry [Koo94; SB06].

3.7 Bounds checks

Like many high-level programming languages and runtimes, the Stack VM specification requires that stack underflow and overflow are caught so that control can return to the surrounding C/C++ environment without crashing the device or performing undefined behaviour. The Stack interpreter

[8]Thumb instructions permit this for addition, subtraction, comparison, and shift operations

implemented alongside the compiler performed a *bounds check* before interpreting each instruction to verify that there are a) enough values on the stack for the instruction's input and b) enough space on the stack to push the results. The compiler could generate equivalent code, but this would incur a significant cost on each instruction. [9] In order to reduce the frequency of stack checks the compiler therefore only emits stack checks at the beginning of basic blocks, as if a stack underflow/overflow *would* occur at some point in a basic block, it is acceptable to halt at the beginning of the basic block with an error, even though the side effects may not be exactly the same.

In order to eliminate the need to load the base and end addresses of the stack from memory, the base and end addresses are always stored in two of the upper general purpose registers; there is a compiler option to disable this.

3.7.1 Bounds check elimination

Many high-level languages support automatic bounds checks on array accesses, but in many cases the checks are redundant and can be eliminated [AT+98; Mos16; QHV02; WWM09]. For instance, in Listing 3.1 there is no need to perform a bounds check on the access `array[i]` as the loop's condition requires that $0 \leq \texttt{i} < \texttt{array.length}$.

```
for (int i = 0; i < array.length; i++) {
    x += array[i];
}
```

Listing 3.1: Example Java loop

There are no arrays in Stack beyond the operand stack itself, so bounds check elimination can only applied to the bounds check that appears at the beginning of each basic block.

The number of values that a basic block accesses already on the stack, the number that it pushes onto the stack, and the offset of the stack pointer from its original value at the beginning of a basic block can all be determined during static analysis.[10] Note that an instruction doesn't necessarily need to 'push' or 'pop' values on the stack in order to access them; the `SWAP` instruction requires that there are two values on the stack already but it doesn't affect the stack pointer.

Let a be the *maximum* number of values accessed on the stack above the initial value of the stack pointer, p the *maximum* number of values pushed to the stack beyond the original stack pointer, and n the offset from the original stack pointer at the end of a basic block. For example, a block of just `ADD` will have $a = 2$, $p = 0$, and $n = 1$, whilst the instruction `DUP` will have $a = 1$, $p = 1$, and $n = -1$.[11] Note that $|n| \leq p$ and $|n| \leq a$.

[9]A comparison and conditional branch are at minimum 4 cycles, but extra arithmetic is required on the stack pointer. The worst case in my compiler for a single check is 11 cycles.

[10]The exception is if instructions like `NDUP`, `NTUCK`, and `NROT` are used with non-constant values. This doesn't matter as these are presently implemented as a function call that itself will perform a bounds check.

[11]Recall that the stack is descending; the stack pointer decreases as values are pushed onto the stack.

After executing a basic block we know that there will be $\geq a - n$ values still on the stack, and the stack still has $\geq p + n$ space before it overflows. Therefore if a basic block jumps to another basic block that accesses $\leq a - n$ elements or pushes $\leq p + n$ values then at least one, if not both, of the stack checks can be skipped. This is surprisingly common; loops will generally leave the stack in a consistent state on both entry and exit. Section 4.2.4 discusses the effect of bounds check elimination on performance.

There are many other cases in which bounds checks can be eliminated, or the checks for several basic blocks can be grouped together. However, most of these techniques require the construction of control flow graphs for functions, which is not viable with the restricted memory of the micro:bit.

3.8 Lazy compilation

As described in Section 3.2, variable destination function calls are permitted, but variable destination branches are not. Variable destination jumps are not a common pattern in any major programming language, but variable function calls are common; object oriented languages use them for dynamic dispatch, for example.

When a call to an unknown location occurs, a call is instead made to an Assembly function that is compiled as part of the core compiler. It looks up in a table to see if the function has already been compiled, and if it has then it jumps to the appropriate destination. This operation is relatively inexpensive, and comparable to the number of instructions/cycles required for indirect calls in C++. On the other hand, if the function hasn't already been compiled then control must return to the compiler so that it can compile the function, along with any other functions that it may call; this works by initiating the static analyser from the function to be compiled rather than the 'main' function. It is possible that the compiler could fail (e.g. if the 'single interpretation rule' discussed in Section 3.2.1 was broken), and if this occurs then Stack code execution is terminated.

When the compiler emits an instruction it appends it to the end of a dynamically-sized array.[12] If this array reaches capacity it will be reallocated, which means that the location of the calling code can change. This isn't a problem for branch instructions, as these use relative offsets, but it is a problem for function return addresses already on the call stack. Therefore a separate piece of Assembly determines if the code has been moved, and if it has then it traverses the call stack and adjusts all return addresses appropriately.

[12]Specifically, a C++ `std::vector`

3.9 Tail Call Optimisation

The compiler supports Tail Call Optimisation (TCO), so if function calls itself and immediately returns then an unconditional branch is performed instead. The interpreter doesn't perform any kind of TCO and instead recurses into itself on function calls. As discussed in Section 4.2.2, the cost of function calls in the interpreter is much greater than in compiled code, so the performance difference for programs that rely on tail recursion is significant.

Chapter 4

Testing

4.1 Correctness

An obvious goal of this project is to ensure that the compiled code behaves correctly. To achieve this over 400 tests were implemented, each consisting of a piece of Stack code and its expected effect on the environment. The majority of the tests were automated and output their success state over a serial link. Tests were broken down into groups that verified

- Instruction encoding produced the expected bit patterns;
- The behaviour of the encoded instruction by itself was as expected;
- Arm instruction decoding produced the expected results;
- The static analyser discovered all reachable basic blocks and functions and annotated them correctly;
- Optional instructions were compiled so that they had they correct effect on the external environment (e.g. turning on LEDs);
- Individual instructions correctly affected the operand stack, regardless of whether values were pushed on before the instruction or during the test;
- The register allocators still produced correct results, even when presented with the opportunity to aggressively optimise generated code;
- Larger programs produced the correct results;
- Larger programs compiled from a high-level language to Stack bytecode executed correctly;[1]

[1]This compiler was developed by Mark Riley, another student

- Stack overflow and underflow were caught caught correctly.

Tests on Stack code were also executed in the interpreter to verify that the behaviour of the compiled code was the same. Appendix D lists the sample code for some of these tests.

4.2 Performance

Performance tests were used to verify that JIT compiled code performed faster than interpreting the same code; this was true in all cases. The process of compilation typically took 10 times longer than interpreting the code, but this was significantly lower in the case of long-running programs.

Two approaches were used to determine the performance of the generated code. The first was a qualitative approach during development by keeping track of the total cycle count of a series of instructions, based on the data presented in Appendix B.[2]

The second, quantitative approach used a Salae Logic Analyser to record bit patterns written over a digital pin during execution. The logic analyser connects to a single digital pin and the ground pin of the micro:bit and records the value of the digital pin — either 0 when its voltage is the same as the ground or 1 when it is higher — at a rate of 12 MHz. Note that this is slightly below the clock rate of 16 MHz of the micro:bit's Cortex-M0 processor. It is therefore not possible to measure with single cycle precision, but this was never needed.

A simple protocol was developed to send short bit patterns over a digital pin. When a 'zero' needed to be sent the digital pin was raised from low to high for a period of 2 cycles, and when a 'one' needed to be sent the digital pin was raised from low to high for a period of 10 cycles. These values are sufficiently high that they will always be detected by the lower precision logic analyser.

The following bit patterns were used:

Bit pattern	Meaning
`00`	Start of interpreting code
`01`	End of interpreting code
`10`	Start of execution of JIT compiled code
`11`	End of execution of JIT compiled code

When each code test executed these bit patterns were issued over the digital pin and the name of the test was printed over the USB serial link. A straightforward Python script took the digital pin values, calculated the time periods between the start and end of interpreting, compiling, and executing the code, and matched these time measurements with the name of each test.

[2] I deliberately implemented the instruction encoder — and all of its tests — for every Thumb instruction before embarking on the rest of the compiler to ensure that I had a good awareness of the performance of individual instructions and the availability of alternative approaches.

These performance tests were designed for comparison of two kinds of performance. Firstly, I wanted to compare the performance of interpreting the code with executing the compiled code. In order to do this, each test was executed 5 times and the total amount of time for setting up the test was subtracted from the time measurement.[3]

Secondly, in order to compare the benefit of certain compiler optimisations the number of cycles per test was computed. Again, this was done by executing each test 5 times but also by subtracting the total amount of time taken to execute the empty program.[4] Note that limitations of the logic analyser mean that some of these values are non-integral.

4.2.1 Microbenchmarks

In microbenchmarks of fewer than 10 Stack instructions, compiled code typically performed 1.4-3.9 times better than interpreting the code. Sample code for these tests can be found in Appendix D.1.[5]

Name	Interpreted code time /μs	Compiled code time /μs	Ratio
AdditionTest	73	46	1.6
AdditionTestPush	164	46	3.6
SubtractionTest	74	46	1.6
SubtractionTestPush	165	46	3.6
MultiplicationTest	74	46	1.6
MultiplicationPushTest	165	46	3.6
DivTest	113	84	1.4
DivPushTest	205	86	2.4
ModTest	107	77	1.4
ModPushTest	199	79	2.5
IncTest	71	45	1.6
DecTest	72	45	1.6
MaxTest	90	48	1.9
MinTest	90	48	1.9
MaxPushTest	182	47	3.9
MinPushTest	182	47	3.9
LtTest	74	47	1.6
LtPushTest	166	46	3.6
DropTest	73	46	1.6

Table 4.1: Single instruction microbenchmarks

[3]For example, it is necessary to ensure the operand stack is empty before executing each test. It only took around 100 cycles to perform this setup, but this value is fairly substantial when tests often executed in far fewer cycles.

[4]There is a small overhead to entering JIT code that only makes sense to remove when comparing JIT compiled code with other JIT compiled code; when comparing interpreted and compiled code this overhead *should* be included because the interpreter also has some overhead for entry

[5]All data is presented rounded to 3 significant figures, but performance ratios are rounded to 2 significant figures.

Tests labelled with ‘Push’ have to firstly perform two push instructions and then perform the operation. The interpreter must perform these operations, but the *compiler* rather than the *compiled code* can perform arithmetic operations with known constants. Therefore in these tests all the compiled code is actually doing is placing some known value on the stack. These tests are not sufficient to make a claim about the expected performance of the compiled code, but they provide a lower bound for the expected performance improvement of JIT compiled code over interpreted code.

More substantial improvements are seen when executing a series of similar instructions, using the stack manipulation instructions `ROT`, `SWAP`, and `TUCK`, or performing conditional branches. The predominant reason for this is that duplicating a value at the top of the stack can typically compile to only a single instruction, pushing a value to the stack can usually be optimised to no more than two instructions, and stack manipulation instructions can be elided when values are already in registers. For instance, ‘RotArithmeticTest’ performs a series of duplications, followed by an alternating series of stack rotations and addition operations. The interpreter executes all of these operations on actual memory, whilst the compiled code avoids executing them at all.

Name	Interpreted code time /μs	Compiled code time /μs	Ratio
DupTest	72	47	1.6
DupManyTest	455	60	7.6
SwapTest	73	47	1.6
RotTest	108	48	2.3
TuckTest	108	48	2.3
RotArithmeticTest	814	52	15.8
RotPushArithmeticTest	259	47	5.6
Push8ManyTest	744	58	12.9
JumpTest	167	49	3.4

Table 4.2: Stack manipulation microbenchmarks. See Appendix D.2.

Instructions for conditional jumps perform significantly better in the compiled code than the interpreted code, as the exact behaviour of comparison instructions doesn’t have to be replicated, so long as the semantics of conditional brnaches remain the same. ‘EqCjmpTest1’, listed in Appendix D.3, performs a comparison followed by a conditional brnach. If the branch is taken this series of instructions takes only 4 cycles. Therefore the lower bound of 6.3 times faster for the execution of conditional jumps in this style suggests that there will be a significant improvement for typical programs involving ‘if’ statements and ‘while’ loops.

Name	Interpreted code time /μs	Compiled code time /μs	Ratio
JumpTest	167	49	3.4
CjmpTest(false)	213	55	3.9
CjmpTest(true)	167	50	3.4
EqCjmpTest1	308	49	6.3

Table 4.3: Branching tests

Overall, these figures suggest that a typical program should therefore perform at least 1.5 times faster when compiled than interpreted, but that we should expect larger improvements (> 4-10 times faster) when jumps and comparisons are involved.

4.2.2 Representative programs

Tests on small numbers of instructions are useful for verifying the correct behaviour of the compiler and finding lower bounds for performance improvement, but they do not serve as an adequate guide for the performance of typical programs, which can benefit from bounds check elimination, as discussed in Sections 3.7.1 and 4.2.4. The programs listed in Appendix D.4 were used to verify the behaviour and performance of more complex Stack programs. These programs demonstrate far more considerable performance improvements than the microbenchmarks.

Name	Interpreted code time /μs	Compiled code time /μs	Ratio
FunctionTest	252	51	5
BoundedRecursionTest	717	74	9.7
IterativeFibonacciTest	6,530	122	53.6
RecursiveFibonacciTest	791,000	21,000	37.7
JumpToFunctionTest	2,190	112	19.6
JumpToNonRecFunctionTest	1,480	82	18
GCDTest	7,320	636	11.5
TailRecTest	3,520	141	25
EnumTest	2,800	297	9.4
LoopExitTest	23,200	2,610	8.9

Table 4.4: Larger program tests

The most significant performance improvements are seen in the Fibonacci programs, as they perform a large number of conditional jumps, which are substantially faster in compiled code than they are in the interpreter. Tests that perform a large number of function calls are also much faster; the interpreter implements these by recursively calling itself, which incurs significant overhead.

The last two tests listed in Table 4.4 are both performed on tests using code generated by another student's (Mark Riley) compiler. Each time that he wanted to update the value of a local variable in his code he generated a call to a function, listed in Listing 4.1, that performed a series of stack rotations to update the corresponding value on the operand stack. The JIT compiler did not implement function inlining, but a future version of the compiler could choose to inline this function. Combined with the existing register allocator, this could eliminate the need to perform *any* operations beyond saving the value to the appropriate memory location at the end of the basic block.

```
DUP INC NROT DROP DEC NTUCK RET
```

Listing 4.1: The `varReplace` function

4.2.3 Register allocation

Section 3.6 describes 3 approaches for register allocation. Note that certain optimisations, such as improving the performance of conditional jumps, are only available with the final register allocator.

Allocator	Naive	Register		Register with CoW	
Test name	Cycles	Cycles	× faster	Cycles	× faster
MaxTest	65	25	2.6	25	2.6
SwapTest	22	24	0.9	22	1
RotTest	122	29	4.2	26	4.7
TuckTest	120	29	4.2	26	4.6
RotArithmeticTest	305	63	4.9	38	8
RotPushArithmeticTest	139	30	4.7	22	6.4
BoundedRecursionTest	109	109	1	109	1
IterativeFibonacciTest	4,170	1,790	2.3	263	15.8
RecursiveFibonacciTest	114,000	98,700	1.2	67,100	1.7
JumpToFunctionTest	325	301	1.1	230	1.4
JumpToNonRecFunctionTest	230	207	1.1	136	1.7
GCDTest	3,800	2,280	1.7	1,910	2
TailRecTest	531	479	1.1	324	1.6
MarksEnumTest	933	892	1.1	825	1.1
MarksLoopExitTest	8,840	8,630	1	8,240	1.1

Table 4.5: A comparison of each register allocation approach

These data reveal that implementing the more advanced register allocators was worthwhile, even though their structure was significantly more complex than the naive allocator. Using the basic 'register' allocator performs better in all tests than the 'naive' allocator, except in certain cases where the 'naive' allocator had best-case implementations already available to it (the 'register' allocator adds an extra redundant store instruction — which costs two cycles — in the implementation of the `SWAP` instruction).

In all cases the register allocator with the copy-on-write mechanism performs much better than the naive allocator.[6] Some of this improvement is the result of performing arithmetic at compile time or generating faster code for conditional jumps, but importantly this allocator avoids redundant loads and stores.

[6]Including all tests not listed here.

4.2.4 Bounds checks and their elimination

Including bounds checks to prevent stack overflow and underflow is expensive, but they ensure that even incorrect programs halt and return to the executing environment safely. Section 3.7.1 discusses conditions under which it is safe to eliminate bounds checks on the stack. Table 4.6 compares the number of cycles executed for different bounds check approaches.

	Cycles			
Name	No checks	Bounds checks	BCE	Benefit of elimination
FunctionTest	20	36	36	0%
BoundedRecursionTest	73	109	109	0%
IterativeFibonacciTest	231	369	263	28.7%
RecursiveFibonacciTest	47,400	77,000	67,100	12.8%
JumpToFunctionTest	110	236	230	2.5%
JumpToNonRecFunctionTest	56	137	137	0%
GCDTest	1,590	1,910	1,910	0.0%
TailRecTest	108	324	324	0%
EnumTest	764	857	825	3.7%
LoopExitTest	7,370	8,370	8,240	1.6%

Table 4.6: Bounds check elimination

Bounds checks are clearly a costly addition to many programs; they make 'TailRecTest' three times slower, for example.[7] Future work could extend the eliminator to remove more checks.

4.2.5 The cost of lazy compilation

To support function calls to locations that are determined dynamically at runtime, the compiler has to support lazy compilation of functions that weren't necessarily discovered in the first round of static analysis. When a function that hasn't been previously compiled is called, the compiler must be re-entered and compilation must start again. The time spent in the compiler depends on the complexity of the code that needs to be compiled.

Dynamic calls	Lazy compilation required /cycles	Compilation done /cycles	$\times$ faster
1	17,100	291	58.9
2	34,500	543	63.4

Table 4.7: Lazy compilation of dynamic calls

Table 4.7 shows that in a single execution both these tests perform substantially worse than the interpreter (the interpreter is about 5 times faster for each program) but once the dynamically called functions have been lazily compiled they then execute in far less time (and significantly outperform the interpreter).

[7]Bounds checks on *some* tail calls can be eliminated, but this particular program doesn't admit any

4.3 Code density

Rather than JIT compiling the bytecode for an arbitrary VM, this project could have received AOT compiled Arm bytecode via serial and linked it on the device. However, Table 4.8 shows that Stack bytecode is generally able to achieve significantly better code density for representative programs.

Name	Stack bytecode size /B	Compiled Arm bytecode size /B	Ratio
FunctionTest	8	132	16.5
BoundedRecursionTest	25	226	9.0
IterativeFibonacciTest	40	234	5.9
RecursiveFibonacciTest	30	202	6.7
JumpToFunctionTest	26	222	8.5
JumpToNonRecFunctionTest	16	152	9.5
GCDTest	27	226	8.4
TailRecTest	19	168	8.8
EnumTest	178	670	3.8
LoopExitTest	280	1086	3.9
LongCodeTest	1004	444	0.44

Table 4.8: Comparison of bytecode sizes

‘LongCodeTest’ is a specially constructed test, described in Appendix D.4, that was used to determine the maximum size of a program that can be compiled. It exploits features of the register allocator so that the Arm bytecode emitted is actually shorter than the original code. Importantly, note that this meets the requirement that the compiler can compile Stack programs of at least 500 B on the micro:bit.

Chapter 5

Conclusion

This project successfully developed a JIT for the BBC micro:bit, alongside software to support programming the micro:bit with Stack programs over serial. The JIT compiler produces code that performs significantly better than interpretation, even though the limited resources of the micro:bit restrict the compiler techniques that could be employed. The successful use of a JIT compiler on such a low-powered device suggests that embedded software for IoT devices could use JITs for deploying software.

5.1 Reflection

I really enjoyed implementing this project, especially the low-level programming. Managing the interoperation of Assembly, C, and C++ was certainly an challenge, but an enjoyable one nonetheless.

I got started with the project much faster than I expected. Some microcontrollers use a *pure* Harvard architecture, which requires program and data memory to be completely separate. The Cortex-M0 uses the more modern *modified* Harvard architecture, which does allow them to be in the same address space. My supervisor and I were initially concerned that it would be necessary to write the generated code to flash before executing it, but resources on self-modifying Arm code revealed this wasn't necessary.

The majority of the instruction encoding code was written in Michaelmas, the bulk of the compiler was written over the Christmas vacation, and I spent Hilary tidying up the compiler and implementing serial transfer.

Whilst I didn't mind implementing the code generation parts of the compiler, I particularly disliked implementing the linker, the code for lazy compilation (because of the need to correct return addresses on the call stack), and some of the later register allocators. I'm a little disappointed that I didn't implement a more advanced register allocator or bounds check eliminator, but limitations of the device ensured that it was unlikely that I'd implement these features. By the end of the project I frequently hit the RAM ceiling of the micro:bit.[1]

Profiling the micro:bit was easier than I expected thanks to the Salae Logic Analyser provided by Alex Rogers, my supervisor. It came with a piece of software that could export the timing data directly to a CSV file. I then wrote a separate set of Python scripts that processed this data for presentation in this report.

5.2 Future work

The JIT compiler presented is portable to any Arm processor that supports Thumb 2 instructions, which includes all ARMv6+ cores. Therefore the compiler could be adapted to other Arm micro-controllers, or indeed most smart phones. Many IoT devices use low-powered wide-area networks for sending and receiving small messages, on the order of less than 100 B per day. Stack's compact representation would allow for transmitting small programs, or fragments of programs, using such protocols to provide software updates for devices deployed "in the field." A future project would use the JIT compiler on an IoT device to support receiving software updates or alternatively create dynamically reconfigurable sensors. The Stack bytecode representation could also be optimised so that more frequently used instructions are encoded in fewer bits, or common combinations of instructions could be reduced to a single higher-level instruction.

An alternative project could also adapt the JIT compiler to support VMs other than Stack. MicroPython is a widely used for programming the micro:bit in schools [GB18]. Existing desktop implementations of Python, such as CPython and PyPy, use a stack-based VM internally [Bol+09; Pyt18]. An adaptation of this project would instead support the internal bytecode of an existing Python implementation, or add JIT compilation support to the MicroPython project.

As described in Chapter 3, a number of traditional JIT techniques were not implemented because of the limited power of the Cortex-M0. As Thumb code can be executed by most Arm processors, the JIT compiler could incorporate more advanced ideas, including tracing. A tracing JIT could also be used to selectively 'uncompile' functions that are used infrequently and instead opt to interpret them, which would free up memory for functions that are used more frequently. Peephole optimisation — which replaces short sequences of instructions with shorter instructions — was also not implemented as it typically requires 'pattern databases' that are often hundreds of kilobytes to several megabytes. With only a few kilobytes of RAM and around 50 kB of flash storage free at the end of the project, pursuing peephole optimisation wasn't viable, even though it typically reduces the code size by 50% [App98; Lam80].

[1]As described in Section 2.1, although there is 16 kB of RAM, only about 8 kB can be user-allocated at runtime.

Appendices

Appendix A

The Stack Virtual Machine

Please note that this appendix is a summary of Alex Roger's original description of Stack [Rog17a].

The Stack Virtual Machine maintains an *operand stack* and *call stack*. Unless otherwise specified, all instructions only affect the operand stack. The program code and data is encoded as an array of bytes, and the virtual machine permits branching to any byte within the array.[1] As instructions are limited to manipulating the operand stack and cannot write back to the code/data array it is not possible to produce self-modifying Stack code. The operand stack is filled with 32-bit signed integers, whilst the call stack uses 32-bit unsigned integers.[2]

If execution reaches the end of the code/data array without halting, execution should terminate as if a halt instruction had been reached.

In the following $P — Q$ denotes a stack effect, where P is the prior state of the top of the stack and Q is the result after popping off P and pushing Q. In both the stack state should be read from right-to-left, i.e. the top of stack is at the end. The result of a boolean operation is either 1 or 0.

The operand values of the `PUSH` instruction are encoded in two's complement little-endian.

[1]This array of bytes is always accessible through the `FETCH` instruction

[2]As these just represent the locations of Stack code they are offsets within the byte array. However, as this stack cannot be directly manipulated through Stack instructions the contents don't actually matter — in the JIT implementation the values correspond to the locations of compiled Arm bytecode.

A.1 Core instructions

Instruction	Encoding (hex)	Effect	Notes
`ADD`	`00`	$ab — x$	$x = a + b$
`SUB`	`01`	$ab — x$	$x = a - b$
`MUL`	`02`	$ab — x$	$x = a * b$
`DIV`	`03`	$ab — x$	$x = a + b$
`MOD`	`04`	$ab — x$	$x = a/b$
`INC`	`05`	$ab — x$	$x = a + 1$
`DEC`	`06`	$ab — x$	$x = a - 1$
`MAX`	`07`	$ab — x$	$x = \max(a, b)$
`MIN`	`08`	$ab — x$	$x = \min(a, b)$
`LT`	`09`	$ab — x$	$x = a < b$
`LE`	`0A`	$ab — x$	$x = a \leq b$
`EQ`	`0B`	$ab — x$	$x = a = b$
`GE`	`0C`	$ab — x$	$x = a \geq b$
`GT`	`0D`	$ab — x$	$x = a > b$
`DROP`	`0E`	$a —$	
`DUP`	`0F`	$a — aa$	
`NDUP`	`10`	$n — x$	x is the n^{th} value on the stack (from zero)
`SWAP`	`11`	$ab — ba$	
`ROT`	`12`	$abc — bca$	
`NROT`	`13`	$n — P$	P corresponds to the remaining top n elements rotated from bottom to top
`TUCK`	`14`	$abc — cab$	
`NTUCK`	`15`	$n — P$	P corresponds to the remaining top n elements rotated from top to bottom
`SIZE`	`16`	$— k$	k is the number of elements on the operand stack
`NRND`	`17`	$n — x$	$x \in \{0, 1, \ldots, n-1\}$
`PUSH`$_8$	`18 x`	$— x$	
`PUSH`$_{16}$	`19 l h`	$— x$	l are the lower 8 bits and h are the upper 8 bits of the 16-bit two's complement representation of x
`FETCH`	`1A`	$a — x$	x is the 16-bit two's complement little-endian at address a in the code/data array
`CALL`	`1B`	$a —$	Pushes the program counter to the call stack and jumps to the address a
`RET`	`1C`	$—$	Pops the return address from the call stack to the program counter
`JMP`	`1D`	$a —$	Jumps to the address at a
`CJMP`	`1E`	$ca —$	Jumps to a if $c \neq 0$
`WAIT`	`1F`	$d —$	Blocks for d ms
`HALT`	`20`	$—$	Halts program execution

A.2 Optional instructions

All optional instructions are encoded in two bytes, where the first byte is always greater than or equal to `80` (in hexadecimal) and the second byte encodes the number of values pushed to the operand stack in the upper four bits, and the number of values popped in the lower four bits. The optional instructions specified below are either from the original description of Stack for the micro:bit, or additions I implemented as part of the project.

Instruction	Encoding (hex)	Effect	Notes
`SLEEP`	`80 01`	d—	Enters low power mode for d seconds and resumes with empty operand and call stacks at address 0.
`TONE`	`81 02`	f—	Starts playing a tone of frequency f, or stops playing if $f = 0$
`BEEP`	`82 02`	fd—	Blocks for d ms whilst playing a tone at frequency f
`RGB`	`83 03`	rgb—	Sets the attached NeoPixel to the colour represented by red r, green g, and blue b
`COLOUR`	`84 01`	c—	Sets the NeoPixel to the colour represented by the three bit colour c
`FLASH`	`85 02`	cd—	Flashes the NeoPixel the 3-bit RGB colour c for d ms (blocking)
`TEMP`	`86 10`	—t	t is the temperature in Celsius
`ACCEL`	`87 30`	xyz—	x, y, z represent the current accelerometer reading on each axis
`PIXEL`	`88 02`	cp—	Sets LED $p \in \{1, 2, \ldots, 9\}$ to colour c
`NUM`	`A0 01`	x—	Uses the micro:bit LEDs to scroll the number x (blocking)

Appendix B

Cortex-M0 instruction cycle counts

The following summarises the cycle counts for the instructions available in the Thumb architecture on the Cortex-M0 that are used by the JIT compiler. The values are as documented by Arm, or in cases where there were ambiguities they have been verified by profiling the nRF51 core on the micro:bit [Arm10].

Operation type	Cycle count
Arithmetic operations (including multiplication) not affecting the PC	1
Register-to-register move not affecting the PC	1
Arithmetic operations or moves affecting the PC	3
Logical and comparison operations	1
Load/store (general)	2
Load/store (n registers)	$1 + n$
Push/pop n registers (excl. PC)	$1 + n$
Pop n registers (inc. PC)	$4 + n$
Conditional branch	3 if taken, 1 if not
Unconditional branch	3
Branch with link	4
Branch with exchange	3
Branch with link and exchange	3

Appendix C

JIT compiler options

The compiler and environment can be configured with a number of different options that affect the performance and size of generated code, as well as a number of debugging options:

- **Allow PC relative loads:** ARM supports loading a 32-bit integer from a fixed offset from the current program counter. The benefit of doing this is that it only takes 2 cycles, and is therefore usually the fastest way to load a large constant (smaller constants can be loaded to a register via a single `mov` instruction). However, this introduces a small amount of extra complexity to the linker, so it is sometimes best avoided, and is disabled by default.

- **Always print compilation:** A debugging mode (off by default) that prints a textual representation of the generated Thumb instructions over the serial link.

- **Bounds check elimination:** Described in Section 3.7.1.

- **Conditional branch type:** The naive approach means that if a comparison instruction (<, <=, =, >=, >) is followed by a `CJMP` then it will generate comparison code that writes a 0 or 1 to the top of stack, and then the `CJMP` code has to compare this value in order to perform the jump. An improved mode, which significantly improves performance for code with a large number of conditional jumps, is on by default and reduces this sequence of instructions to a single comparison followed by a single conditional jump.

- **Profiling enabled:** Ensures that special bit patterns are sent over one of the micro:bit's digital output pins before and after execution of interpreted or compiled code. Used for performance testing. Off by default.

- **Register allocation:** Switches between the three approaches discussed in Section 3.6.

- **Register write elimination:** adopts the approach of allowing separate read/write register maps, as discussed in Section 3.6.3. On by default.

- **Stack check mode:** Can be used to disable bounds checks, which help to prevent stack overflow and underflow. The default approach generates a small amount of code (< 10 instructions) at the beginning of each basic block, but there is the alternative approach of performing a function call to test for stack underflow or overflow. This performs slightly worse in the case that stack checks usually succeed, but is better in the case when they usually fail. As this is not the expectation it is not on by default.

- **Tail calls optimised:** On by default.

Appendix D

Stack test samples

This appendix lists sample test programs that were used to verify the correctness and performance of the compiler. For reference the original Stack assembly and Stack bytecode, encoded in hexadecimal, have been included.

D.1 Single instruction samples

Add test

Two values were pushed to the stack prior to the execution of the test. Similar tests were also used for all other binary operators.

```
ADD         00
```

Add push test

The test itself pushed values onto the stack rather than assuming that they were there already:

```
37          18 25
42          18 2A
ADD         00
```

Again, such tests were used for all binary operators.

D.2 Stack manipulation samples

Dup Many Test

```
DUP         0F
DUP         0F
DUP         0F
...
```

The final version of this test used 10 `DUP` instructions. Several instructions, especially stack manipulation functions that could be elided by the later register allocators, were tested in this manner to ensure their behaviour was still correct.

Rot and Arithmetic Test

```
1           18 01
DUP         0F
INC         05
DUP         0F
INC         05
DUP         0F
INC         05
DUP         0F
INC         05
ROT         12
ROT         12
ADD         00
ADD         00
ADD         00
ADD         00
ADD         00
```

This test was used for checking correctness when the compiler could completely eliminate arithmetic operations, checking correctness of the register allocator when `DUP` instructions were used, and comparing the performance of the more advanced register allocators.

D.3 Conditional Jump sample

The following test begins with two values already pushed onto the stack, and pushes different values depending on whether they are equal.

```
  EQ        0B
  eq_dest   18 07
  CJMP      1E
  10        18 0A
  HALT      20
eq_dest:
  20        18 14
```

D.4 Larger program samples

Function test

Calls a function to multiply two numbers on the top of stack together.

```
  func CALL HALT    18 04 1B 20
func:
  2 MUL RET         18 02 02 1C
```

Iterative Fibonacci

Computes the 18th Fibonacci number using an iterative algorithm.

```
  18 fibonacci CALL     18 12 18 06 1B
  HALT                  20
fibonacci:
  DUP                   0F
  1 GT isGtOne CJMP     18 01 0D 18 0E 1E
  RET                   1C
isGtOne:
  0 1                   18 00 18 01
loop:
  DUP TUCK              0F 14
  ADD                   00
  ROT                   12
  1 SUB DUP 4 NTUCK     18 01 01 0F 18 04 15
  1 GT loop CJMP        18 01 0D 18 12 1E
  ROT DROP SWAP DROP    12 0E 11 0E
  RET                   1C
```

Recursive Fibonacci

This program computes the 15th Fibonacci number using a recursive algorithm.[1]

```
  15 fibonacci CALL     18 0F 18 06 1B
  HALT                  20
fibonacci:
  DUP                   0F
  1 GT gtOne CJMP       18 01 0D 18 0E 1E
  RET                   1C
gtOne:
  DUP                   0F
  1 SUB fibonacci CALL  18 01 01 18 06 1B
  SWAP                  11
  2 SUB fibonacci CALL  18 02 01 18 06 1B
  ADD                   00
  RET                   1C
```

Greatest Common Divisor

The greatest common divisor of two numbers is computed using the standard tail recursive algorithm.

```
  610 987 gcd CALL      19 62 02 19 DB 03 18 0A 1B
  HALT                  20
gcd:
  DUP                   0F
  0 EQ ret0 CJMP        18 00 0B 18 19 1E
  DUP ROT SWAP MOD      0F 12 11 04
  gcd CALL RET          18 0A 1B 1C
ret0:
  DROP RET              0E 1C
```

[1]This program is not intended to compare the performance of the iterative and recursive algorithms; it doesn't matter that they computer different Fibonacci numbers

Summary of other larger programs

- **BoundedRecursionTest:** Perform a series of 6 recursive calls to further demonstrate that function calls are less expensive in compiled code.
- **JumpToFunctionTest:** Perform a while loop that decrements a value to zero where the start of the while loop is at the same address as the start of a function.[2]
- **JumpToNonRecFunctionTest:** Similar to the above.
- **TailRecTest:** Use a tail-recursive function to decrement the top of stack down to 0, starting at 10. In the interpreter this requires 10 recursive calls, but in the tail-call-optimised compiled code this is reduced to code equivalent to a while loop.
- **EnumTest:** Based on compiled code generated by Mark Riley's compiler for a high-level language to Stack bytecode. This included a number of patterns, such as using a function to treat specific values on the operand like variables, that could be optimised by features of the compiler including tail-call-optimisation and the register allocator.
- **LoopExitTest:** Also generated by Riley's compiler. This program had several nested loops in a high-level language, and translated to Stack bytecode containing a lot of unconditional and conditional jump instructions.
- **LongCodeTest:** The test initially assumes a value n on the stack. It then pushes a onto the stack, and repeatedly performs the sequence of instructions:

  ```
  1 ndup rot add swap
  ```

  These leave the values a and $n + a$ on the operand stack. Therefore if the above is repeated k times, the final values on the stack are a and $n + ka$. This is clearly an inefficient way to do this computation; the test was used to determine the maximum bytecode size that can be compiled.

[2]This test was originally developed to ensure the correctness of the linker.

Bibliography

[Aho+06] A. V. Aho et al. *Compilers: Principles, Techniques, and Tools*. 2nd ed. 2006. ISBN: 0-201-10088-6.

[App98] A.W. Appel. *Modern Compiler Implementation in ML*. 1st ed. Cambridge University Press, 1998.

[Arn+04] M. Arnold et al. *Architecture and Policy for Adaptive Optimization in Virtual Machines*. 2004. URL: https://researcher.watson.ibm.com/researcher/files/us-hindm/RC23429.pdf.

[AT+98] A. Adl-Tabatabai et al. "Fast, Effective Code Generation in a Just-in-time Java Compiler". In: *SIGPLAN Not.* 33.5 (May 1998), pp. 280–290. ISSN: 0362-1340. DOI: 10.1145/277652.277740. URL: http://doi.acm.org/10.1145/277652.277740.

[Bol+09] C. F. Bolz et al. "Tracing the Meta-level: PyPy's Tracing JIT Compiler". In: *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ICOOOLPS '09. Genova, Italy: ACM, 2009, pp. 18–25. ISBN: 978-1-60558-541-3. DOI: 10.1145/1565824.1565827. URL: http://doi.acm.org/10.1145/1565824.1565827.

[Bra13] J. Bramley. *Caches and Self-Modifying Code*. 2013. URL: https://community.arm.com/processors/b/blog/posts/caches-and-self-modifying-code.

[Ert96] M. A. Ertl. "Implementation of Stack-Based Languages on Register Machines". PhD thesis. Technische Universität Wien, Austria, Apr. 1996.

[Ert99] M. A. Ertl. *Forth FAQ*. 1999. URL: http://www.forth.org/faq/faq1.txt.

[FMM16] J. Finney, R. May, and M. Moskal. *Metadata for partial flashing*. 2016. URL: https://github.com/lancaster-university/microbit-dal/issues/206.

[GB18] D. George and BBC. *MicroPython for the BBC micro:bit*. 2018. URL: https://github.com/bbcmicrobit/micropython.

[Gre18] M. Grebe. *Haskino*. 2018. URL: https://github.com/ku-fpg/haskino.

[Haw16] M. Hawkins. *Using the BBC Microbit with the Raspberry Pi*. 2016. URL: https://www.raspberrypi-spy.co.uk/2016/05/using-the-bbc-microbit-with-the-raspberry-pi/.

[KG02] A. Krishnaswamy and R. Gupta. "Profile Guided Selection of ARM and Thumb Instructions". In: *SIGPLAN Not.* 37.7 (June 2002), pp. 56–64. ISSN: 0362-1340. DOI: 10.1145/566225.513840. URL: http://doi.acm.org/10.1145/566225.513840.

[Kna93] P.J. Knaggs. “Practical and Theoretical Aspects of Forth Software Development”. PhD thesis. University of Teesside, Mar. 1993.

[Koo94] P.J. Koopman. “A Preliminary Exploration of Optimized Stack Code Generation”. In: *Journal of Forth Applications and Research* 6 (1994), pp. 241–251.

[Lam80] D. A. Lamb. *Construction of a peephole optimiser*. Aug. 1980. URL: http://repository.cmu.edu/cgi/viewcontent.cgi?article=3516&context=compsci.

[Lin+15] T. Lindholm et al. *The Java Virtual Machine Specification*. Feb. 2015. URL: https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf.

[Mos16] A. Mosoi. *Bounds Check Elimination for Go*. 2016. URL: https://docs.google.com/document/d/1vdAEAjYdzjnPA9WDOQ1e4e05cYVMpqSxJYZT33Cqw2g/edit#.

[Pal17] M. Pall. *LuaJIT Performance: ARM*. 2017. URL: http://luajit.org/performance_arm.html.

[Piz16] F. Pizlo. *Introducing the B3 JIT Compiler*. 2016. URL: https://webkit.org/blog/5852/introducing-the-b3-jit-compiler/.

[QHV02] F. Qian, L. Hendren, and C. Verbrugge. “A Comprehensive Approach to Array Bounds Check Elimination for Java”. In: *Compiler Construction*. Ed. by R. Nigel Horspool. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 325–341. ISBN: 978-3-540-45937-8.

[RCM93] E. D. Rather, D. R. Colburn, and C. H. Moore. “The Evolution of Forth”. In: *SIGPLAN Not.* 28.3 (Mar. 1993), pp. 177–199. ISSN: 0362-1340. DOI: 10.1145/155360.155369. URL: http://doi.acm.org/10.1145/155360.155369.

[Rog17a] A. Rogers. “Stack: a stack-based virtual machine for physical devices”. 2017.

[Rog17b] A. Rogers. *Stack IDE*. 2017. URL: http://www.cs.ox.ac.uk/people/alex.rogers/stack/.

[Sap16] S. Sapin. *Rust on the BBC micro:bit*. 2016. URL: https://github.com/SimonSapin/rust-on-bbc-microbit.

[SB06] M. Shannon and C. Bailey. “Global Stack Allocation - Register allocation for Stack Machines”. 2006.

[Sch05] M. Schoeberl. “Design and Implementation of an Efficient Stack Machine”. 2005.

[Sen+16] S. Sentance et al. *micro:bit evaluation report*. Aug. 2016. URL: https://www.kcl.ac.uk/sspp/departments/education/research/Microbit-Evaluation-Report-for-Microsoft.pdf.

[Sen+17] S. Sentance et al. *Creating Cool Stuff - Pupils’ experience of the BBC micro:bit*. English. Mar. 2017.

[Sha06] M. Shannon. “A C Compiler for Stack Machines”. MA thesis. University of York, 2006.

[Shi+08] Y. Shi et al. “Virtual Machine Showdown: Stack Versus Registers”. In: *ACM Trans. Archit. Code Optim.* 4.4 (Jan. 2008), 2:1–2:36. ISSN: 1544-3566. DOI: 10.1145/1328195.1328197. URL: http://doi.acm.org/10.1145/1328195.1328197.

[Vau12] B. Vaugon. *Programming PIC microcontrollers with OCaml*. 2012. URL: http://www.algo-prog.info/ocapic/web/lib/exe/fetch.php?media=ocapic-tutorial-1.2.pdf.

[WWM09] T. Würthinger, C. Wimmer, and H. Mössenböck. “Array Bounds Check Elimination in the Context of Deoptimization”. In: *Sci. Comput. Program.* 74.5-6 (Mar. 2009), pp. 279–

295. ISSN: 0167-6423. DOI: 10.1016/j.scico.2009.01.002. URL: http://dx.doi.org/10.1016/j.scico.2009.01.002.

[Ada18] Adafruit. *Neopixels*. 2018. URL: https://www.adafruit.com/category/168.

[Arm05a] Arm Holdings. *ARM Architecture Reference Manual*. 2005.

[Arm05b] Arm Holdings. *New ARM Jazelle RCT Technology Provides As Much As Three Times Reduction In Java Memory Footprint*. 2005. URL: https://www.arm.com/about/newsroom/9621.php.

[Arm10] Arm Holdings. *ARM Cortex-M0 Processor Technical Reference Manual*. 2010. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.cortexm.m0/index.html.

[Arm12] Arm Holdings. *ARM Cortex-M Programming Guide to Memory Barrier Instructions*. 2012. URL: http://infocenter.arm.com/help/topic/com.arm.doc.dai0321a/DAI0321A_programming_guide_memory_barriers_for_m_profile.pdf.

[Arm14] Arm Holdings. *ARM Architecture Reference Manual for ARMv7, ARMv7-A*. 2014. URL: https://static.docs.arm.com/ddi0406/c/DDI0406C_C_arm_architecture_reference_manual.pdf.

[Arm15] Arm Holdings. *Procedure Call Standard for the ARM Architecture*. 2015.

[Arm17] Arm Holdings. *ARM Architecture Reference Manual for ARMv8, ARMv8-A*. 2017. URL: https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf.

[Arm18a] Arm Holdings. *Arm Cortex-M Series Processors*. 2018. URL: https://developer.arm.com/products/processors/cortex-m.

[Arm18b] Arm Holdings. *ARM CPU architecture*. 2018. URL: https://developer.arm.com/products/architecture/cpu-architecture.

[Arm18c] Arm Holdings. *Cortex-M0*. 2018. URL: https://developer.arm.com/products/processors/cortex-m/cortex-m0.

[Arm18d] Arm Holdings. *Cortex-M0+*. 2018. URL: https://developer.arm.com/products/processors/cortex-m/cortex-m0-plus.

[Atm16] Atmel. *ATmega328/P Datasheet*. 2016. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf.

[BBC15] BBC. *BBC micro:bit: Groundbreaking initiative to inspire digital creativity and develop a new generation of tech pioneers*. 2015. URL: http://www.bbc.co.uk/mediacentre/mediapacks/microbit.

[Fre14] Freescale Semiconductor, Inc. *Kinetis KL25 Sub-Family*. 2014. URL: https://www.nxp.com/docs/en/data-sheet/KL25P80M48SF0.pdf.

[GCC18] GCC. *Options for linking*. 2018. URL: https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html.

[Goo18] Google. *Chrome V8*. 2018. URL: https://developers.google.com/v8/.

[Int17] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 2017. URL: https://software.intel.com/en-us/articles/intel-sdm#optimization.

[Lan16] Lancaster University. *Micro:bit runtime*. 2016. URL: https://lancaster-university.github.io/microbit-docs/.

[LLV18a] LLVM Admin Team. *The LLVM Compiler Infrastructure Project*. 2018. URL: http://llvm.org.

[LLV18b] LLVM Project. *LLD - The LLVM Linker*. 2018. URL: http://lld.llvm.org.

[LoR18] LoRA Alliance. *LoRA Alliance™ Technology*. 2018. URL: https://www.lora-alliance.org/technology.

[Mic16] Micro:bit Education Foundation. *BBC micro:bit*. 2016. URL: http://microbit.org/about/.

[Mic17a] Micro:bit Developer Community. *micro:bit Hardware Description*. 2017. URL: http://tech.microbit.org/hardware/.

[Mic17b] Microsoft. *Common Language Runtime*. Oct. 2017. URL: https://docs.microsoft.com/en-us/dotnet/standard/clr.

[Mic18] Microsoft Research. *Microsoft MakeCode*. 2018. URL: https://makecode.microbit.org.

[Mon18] Mono Project. *ARM*. 2018. URL: http://www.mono-project.com/docs/about-mono/supported-platforms/arm/.

[Nor14] Nordic Semiconductor. *nRF51 Series Reference Manual*. Oct. 2014. URL: https://www.nordicsemi.com/eng/nordic/download_resource/62725/13/67992764/13233.

[Ora18] Oracle. *The Java HotSpot Performance Engine Architecture*. 2018. URL: http://www.oracle.com/technetwork/java/whitepaper-135217.html.

[PyP18] PyPy. *PyPy Speed Center*. 2018. URL: http://speed.pypy.org.

[Pyt18] Python Foundation. *CPython*. 2018. URL: https://github.com/python/cpython.

[Ras18] Raspberry Pi Foundation. *Raspberry Pi*. 2018. URL: https://www.raspberrypi.org/about/.

[Sig18] Sigfox. *Sigfox Technology Overview*. 2018. URL: https://www.sigfox.com/en/sigfox-iot-technology-overview.