A parallel embedded processor and a concurrent concatenative programming language

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

Trinity 2019

Abstract

The computational power available in embedded environments increased significantly in recent years. Higher silicon budgets admit the use of memory protection and multiple cores, allowing for the execution of parallel and concurrent programs in the embedded environment. Despite advances in hardware, these devices are largely programmed with languages designed for the computers of the 1970s.

Following work in the late 20th century, we introduce Stannel, a new processor architecture inspired by the Inmos Transputer. Stannel supports an arbitrary number of cores for parallel programming with channels as its fundamental model for inter-process communication. We present an implementation of Stannel for the low-energy Lattice Semiconductor Field Programmable Gate Arrays (FPGAs).

We also present a compiler for Statick, a new high-level, concurrent, concatenative programming language with a Hindley-Milner type system. Statick directly reflects the capabilities of Stannel, motivating an alternative model for embedded programming.

Acknowledgements

I'd firstly like to thank Professor Alex Rogers, my supervisor, for introducing me to low-level hardware development, processor design, and FPGA development. He taught me to identify many classes of bugs in my design, and without his help Stannel certainly wouldn't function! It's been a pleasure to collaborate with him over the last two years.

Professor Gavin Lowe showed me a brighter path for concurrent programming and Karel Hruda mentioned the Transputer frequently enough over the past four years to pique my curiosity. Our conversations motivated many features in my design, and I learned a great deal from them. They collectively share responsibility for admitting me to Oxford in the first place, and I'm forever grateful for this four year journey of learning and discovery!

Finally, thanks to all my friends and family, especially for putting up with my conversations of Verilog, type checking, and compilers over the last year.

Contents

Chapter 1

Introduction

Through the 1980s and 1990s Inmos sold the Transputer, a microprocessor that supported channels as its low-level communication model between processes [Inm87]. Inmos also designed the Occam programming language [May87] in collaboration with Tony Hoare, the designer of Communicating Sequential Processses (CSP) [Hoa03]. Today, major programming languages promote channels as their primary approach for allowing processes to communicate [Goo19]. Conversely, the Transputer has had comparatively little influence on the design of modern micro-architectures.

This project contributes a new micro-architecture design for embedded processors based on ideas from the Transputer, making changes to suit the micro-architecture for devices with low-energy requirements, and taking advantage of hardware support to efficiently implement high-level architecture features.

Additionally, we reconsider the use of concatenative programming languages for embedded devices, particularly those which support concurrency. We contribute the first statically-typed concatenative programming language with concurrency support and type-based verification of communication operations.

1.1 Motivation

Despite plenty of evolution in hardware, computers are still widely programmed in C, a language that has remained relatively stagnant since the early 1970s. Proponents of C argue that it is 'close-to-the-metal' as most C features correspond directly to small numbers of instructions, but the language does not exploit all architectural features, particularly with regard to concurrency [Chi18]. Furthermore, C's 'low-level' nature allows programmers to easily introduce security vulnerabilities, particularly with respect to memory, that could be detected or prevented in higher-level languages. We therefore seek a new language that is 'low-level' in its operations, but sufficiently 'high-level' that tools can detect common memory and communication errors at compilation time.

In the modern world, we are increasingly surrounded by low-powered, embedded processors. In recent years, ARM designs dominated embedded micro-processor architectures, and these designs are converging on ideas commonplace in desktop architectures. It is reasonable to expect that within the next decade the standard for low-powered embedded processors will encompass processors with multiple cores, virtual memory, and deep pipelines — such processors already exist [ARM18]. Unlike their desktop counterparts, these processors typically support reading and writing from main memory within a few cycles. Comparatively, a modern desktop processor will take hundreds of cycles to interact with main memory and will use complex cache architectures to alleviate this latency.[1] With 'fast' memory, instructions can frequently interact with main memory. With fewer constraints than desktop environments it is possible to explore alternative architectures for embedded processors: we will therefore design a new low-powered, multi-core embedded processor for our compiler to target.

With fast memory, architectures that frequently interact with memory are viable, such as stack-based machines. Stack-based Instruction Set Architectures (ISAs) require fewer instructions and admit compact binary encodings of programs, placing them as strong alternatives to modern RISC architectures in embedded environments. Stack-based ISAs are the intermediate languages for most major Just-In-Time (JIT) compilers [Lin+15; Mic17], but there has been comparatively little development of stack-based hardware architectures in recent years: our processor will be stack-based.

Modern multi-core architectures share resources between cores, such as caches and main memory, but eschew instructions for direct communication between processes and instead favour schemes involving system calls, memory protection barriers, and compare-and-set instructions [Int18]. These are *very* reasonable approaches when considering the deep memory hierarchies of these architectures, but alternative, channel-based approaches are possible with fast memory.

FPGAs offer fertile ground for prototyping new processor designs. Low-powered FPGAs, such as Lattice's iCE series, can be used to prototype processor designs that can be embedded in micro-controllers.

With a stack-based architecture in mind, we are motivated to design a programming language that simplifies stack interaction. Concatenative programming languages compose functions through the binary operation of concatenation; the function on the right takes the output of the left as its input [Thu08]. Concatenative languages typically operate on stacks of values, although they are not required to. Historically, concatenative languages were dynamically typed, making compile-time optimisations and verification challenging. Most concatenative programming languages do not support concurrency or communication amongst their core primitives. There are no existing statically-typed concatenative programming languages with native support for concurrency, but we demonstrate these can improve program safety and simplicity.

Static type systems need not be an intrusive addition to a concatenative programming language; functional programmers have long espoused the benefits of global type inference, allowing the programmer to write statically-typed programs without ever having to declare types of values or functions. Further, languages with Hindley-Milner type inference provide parametric polymorphism, obviating the need for duplicating function definitions when parameter types change. Stack manipulation operations are obvious candidates for parametric polymorphism, along with functions composed of such operations. Affine, linear, and dependent types support constraining resource usage and can enforce compulsory resource use and automatic memory management at no runtime cost. These additions to our type system introduce extra safety to the language whilst adding no burden to the programmer.

This work motivates an alternative approach to programming concurrent, embedded systems by avoiding traditional programming languages and architectures to admit safety and simplicity, whilst maintaining performance.

[1]An Intel i7 processor has 3 caches levels with worst-case access time for L3 on the order of 300 cycles [Fog18; Lev09].

1.2 Requirements

1.2.1 Hardware and Instruction Set

We present a new ISA with instructions for stack manipulation, creating processes, and communication by sending messages on channels. The ISA supports arbitrarily many cores on the low-power Lattice iCE FPGA.[2] The full implementation operates as a System-on-a-Chip (SoC) that receives a program's bytecode via USB, executes the program, and returns the contents of each process's stack to the host computer when no process can be scheduled.

The memory of each process is isolated. The only way for a process to affect the memory of another process is by communicating with it via a channel. The ability to create and destroy processes and channels via a single instruction requires a hardware component for the allocation or deallocation of memory. As processes are managed in hardware, we developed a hardware scheduler for processes.

Each core executes stack manipulation instructions within N cycles, where N is the number of memory operations the instruction may cause. This restriction reflects the capability of the memory available on the Lattice iCE FPGA. In the event of communication between processes, the number of cycles to handle the communication should again be proportional to the number of memory operations required to execute the instruction. For programs executing a single process the total number of cycles is comparable to other low-powered micro-architectures such as the ARM Cortex M0 [Arm10].

Each component of the processor is rigorously and independently tested on a range of reasonable input values to demonstrate it works correctly. A single core's execution of each instruction is tested, and finally the processor is tested as a whole with hundreds of test programs. Additionally, an instruction-level software simulator of the processor is provided, and utilises the same test suite.

1.2.2 Programming Language

The syntax, semantics, and type system for a concatenative, statically-typed language are formally described. The programming language supports a boolean type, an integer type, a parametric channel type, and parametric function types that manipulate the stack. It features functions that send and receive messages on channels and for listening on several channels in an alternation. The standard library of functions directly reflects the hardware operations of the ISA described in Section 1.2.1. The compiler's components are independently tested, and we also measure the performance of the bytecode it produces.

The type system of our language is based on the Hindley-Milner type system, which is the basis of the Haskell type system amongst other languages. This work builds on existing work that adapts Hindley-Milner to concatenative languages [Dig08b; Dig18] by extending it with support for channel operations, and limited forms of affine, linear, and dependent types to ensure safety of communication operations; channels are either 'used forever' or destroyed after a fixed, finite number of uses.

[2]The number of cores present on the final processor is determined by the number of logic units available.

1.3 Challenges and Restrictions

The processor runs on a Lattice iCE FPGA, as it is affordable, uses little power, and was reverse engineered to support an open source toolchain [Wol18a]. The choice of FPGA immediately imposes a number of restrictions on the design. Firstly, each FPGA has 16 KiB of RAM divided into 32 512 B cells [Lat16]. On each cycle, only a single 2 B word can be read or written to each cell. This imposes a reasonable limit of a processor that uses 16 bit words, 8 bit addresses, and allows a single process to address at most 512 B (i.e. one full memory cell). Processes may use up to 16 of these cells, with the remainder unused or dedicated to scheduling and communication.

To admit pipelining of fetching and execution of instructions, programs and data use a separate address space.[3] Further, channels are restricted to communicating a single word from at most one sender to at most one receiver, with communication synchronised between processes. This restriction is reasonable as buffered channels and channels with many senders or receivers can be simulated with processes and alternations with little overhead.

Lattice iCE FPGAs have a maximum clock speed of 100 MHz, but our design restricts the clock to 16 MHz to simplify timing constraints and memory access. This reduced clock speed is comparable to embedded ARM processors.

It was not possible to fully verify the execution of the processor due to the limited capabilities of tools available and amount of time available. In lieu of formal verification, we developed a large number of unit tests that examine the output of each component of the processor under different inputs, in addition to a wide suite of test programs.

Our compiler will not aim to produce fast code, but instead aim to take advantage of the type system to verify that code is correct before execution. We demonstrate some limited optimisations, such as peephole optimisation and dead code elimination. We constrain the language to obviate the need for monomorphisation of polymorphic functions during code generation.

1.4 Contributions

This project contributes Stannel, a multi-core embedded processor based on stack machines along with an implementation on a commodity FPGA.[4] The design supports the deployment and execution of programs from a host computer via USB. The use of channels and extensible design admits the possibility of extending the processor's hardware to introduce new computation units, such as those used in machine learning workflows, without introducing new instructions. We developed a test suite to verify the correctness of the processor.

We also present Statick, the first statically-typed concatenative programming language with support for communication between processes via channels.[5] It utilises a Hindley-Milner based type checker to support global type inference and parametric polymorphism. Channels use properties of affine, linear, and dependent type systems to statically verify that is either always or never possible to communi-

[3] Our architecture is Harvard rather Von Neumann.

[4] Stannel = stacks + channels.

[5] Statick = static typing + stacks.

cate with them. To date, it is the first programming language of any kind to enforce management of communication channels in this manner.

In summary, we present a new micro-architecture for use in embedded devices and its implementation and a novel statically-typed concatenative programming language and its compiler. Together the architecture and language suggest a new approach to programming embedded devices, whilst still using minimal energy.

Finally, we consider future extensions of this project, including modifications to and extensions of Stannel, targeting other architectures in the Statick compiler, and advancing the verification performed by the Statick compiler.

1.5 Outline of the report

Chapter 2 summarises the target FPGA, the Inmos Transputer, concatenative programming languages, Hindley-Milner type systems, and other related work. Chapter 3 describes the design of the Stannel ISA and processor. Chapter 4 discusses the formal semantics of the Statick programming language and its type system. Chapter 5 considers the correctness of the Stannel processor and the Statick compiler. Chapter 6 motivates future work and extensions. The appendices include complete descriptions of the Stannel ISA, the Statick type system, and our type checking algorithm.

Our implementation is available at `https://github.com/thomasdenney/statick-and-stannel`.

Chapter 2

Background

We assume familiarity with CSP, λ-calculus, compilers, and pipelined processors as presented in the Concurrency [GR+16], Lambda Calculus and Types [Ker09], Compilers [Spi18], and Computer Architecture [Rog16] courses.

2.1 FPGAs

FPGAs are reprogrammable circuits widely used in industry for prototyping new integrated circuits. They comprise a variable number of reprogrammable logic cells, which are internally reconfigured to support different combinatorial functions via lookup tables. Cells also include flip-flop registers for sequential logic. After determining the logic function of each cell, a tool routes 'wires' between cells to complete the design. FPGAs may include RAM cells and support for communication over digital pins or USB. Designers use Hardware Descript Languages (HDLs), which abstract logic cell organisation to 'register-transfer-level', to program FPGAs. *Synthesis* compiles HDLs to *bitstreams*, which reprogram FPGAs by describing cell functions and layout.

HDL tools support simulating designs in software, with suites of tools available to inspect the value of each wire or register at each clock step. We developed our processor through a combination of software simulation and testing on an FPGA with Verilog, an HDL.

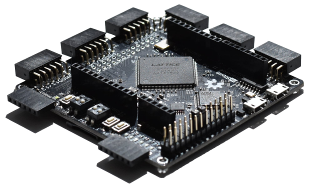


Figure 2.1: The BlackIce II board and Lattice iCE FPG.A

We used the Lattice iCE HX8K FPGA on a BlackIce II board [Lat19; Ver17]. The Lattice iCE series consume little energy and the HX8K has the greatest number of programmable logic and RAM cells in the series. Each RAM cell has a capacity of 512 B. Its clock runs at 100 MHz, but more complex designs cannot run at full speed because values do not propagate along wires fast enough. The BlackIce II board is an open-source prototyping board that features USB programming and communication via the UART protocol. Typically FPGAs must be programmed using proprietary tools, but the iCE series was reversed engineered, and YoSys, an entirely open-source suite of tools that we used for this project, is a popular alternative to Lattice's own tools [Wol18a; Wol18b].

2.2 Occam and the Inmos Transputer

Figure 2.2: Inmos Transputer Evaluation Module in The National Museum of Computing, Bletchley.

The Transputer was an innovative computer sold by Inmos during the 1980s. Unlike most computers of its era, it emphasised concurrent, communicating processes. Processes and channels were first-class concepts in the architecture, with instructions to send and receive messages. This notion was unusual at the time and remains unusual today: the complex memory hierarchies of modern architectures preclude the inclusion of such instructions and it is unlikely that any major desktop architecture will directly adopt such a scheme.

Against the backdrop of the microcomputers of the 1980s, the Transputer was a commercial failure. With Moore's Law still in full swing, Inmos' competitors rapidly increased their simpler designs' clock speeds, whilst the complex Transputer struggled to keep pace. By the end of the decade, Inmos closed down, with its assets sold off [Sel07].[1] Nevertheless, the Transputer had a lasting impact. Many of its instructions were micro-coded, i.e. each user-level instruction decoded into a sequence of simpler instructions executed by a RISC processor, and today Intel's x86 architecture and its successors implement instructions with 'micro-operations' [BO16]. Highly concurrent architectures are now also common-place: GPUs often contain hundreds of compute units.

The Transputer used a stack architecture with three registers arranged in a stack. Instructions collected their operands from these registers rather than encoding them into the instruction itself, unlike other contemporary and modern architectures. Encoding instructions in this manner reduced the overall instruction count (a minimal implementation contained fewer than 30 instructions), but came at a cost: by this point processor performance outpaced RAM performance, and instructions that needed to manipulate both register- and memory-based stacks were costly to execute [Inm87].

Inmos also sold a compiler for Occam, a language designed to take advantage of the Transputer [May87], whilst deriving much of its theory from Tony Hoare's CSP [Hoa03]. It featured channels as its first-class primitive for inter-process communication, and these operations efficiently compiled to instructions for the Transputer. Occam had a lasting impact; major programming languages such as Go and Rust promote channels as the *default* approach for sharing values in a concurrent program [Goo19; Moz18].

2.3 Concatenative Programming Languages

Concatenative programming languages compose functions through concatenation [Con17]. The output of a function in a composed function serves as the input of the next — typically the input and output values are stacks of values, although they do not need to be.[2] Forth is the most prominent example of a concatenative language, and remains popular for embedded systems programming as it makes little abstraction over the internal stack machine of many embedded processors [Bro81].

[1] Inmos employees had a lasting impact on the semiconductor industry: for example, a former Inmos employee co-founded Lattice Semiconductor, the manufacturers of the FPGA we used to design a successor to the Transputer.

[2] Alternatively, functions mutate *the* singular stack.

2.4 Hindley-Milner Type Systems

The Hindley-Milner type system is an extension of the type system of Simply Typed λ-calculus with support for parametric polymorphism, so that the bound term of an abstraction expression may be generic over types, rather than just one fixed type. The authors developed Algorithm J, which is notable for its deduction of an expression's type in linear time in the absence of information beyond an expression's syntax [Hin69; Mil78; DM82].

Outside theoretical computer science, ML and its derivatives, including Haskell and F#, adopted Hindley-Milner-based type systems. Consequently, these languages never require the programmer to annotate functions with types [PJ+98].

2.5 Affine, Linear, and Dependent Types

In the context of extensions to Simply Typed λ-calculus, an *affine* type is one that allows weakening and exchange, and a *linear* type is one that only permits exchange [TP11]. In programming language type systems, affine types ensure that a value has at most one owner (i.e. the value can be discarded), whilst linear types ensure that a value has exactly one owner (the value can never be discarded). The Rust programming language uses affine types to enforce its rules for ownership and sharing of values [Jal17]. Linear types are a proposed extension of Haskell's type system [Ber+17], and there are existing implementations for other programming languages [Bak92].

Meanwhile, dependent types "allow types to be predicated on values." [Bra19] A straightforward example is a matrix multiplication function whose type determines that the function takes an $M \times N$ matrix and a $N \times P$ matrix to produce an $M \times P$ matrix as a result. Idris is a prominent functional language featuring dependent types. Statick, our programming language, uses a limited form of dependent types for channel operations.

2.6 Related Work

Stannel, our processor architecture, takes direct inspiration from the Inmos Transputer. However, we contribute several significant changes to modernise the instruction set, and our architecture aims to support embedded systems programming, rather than a full desktop environment.

The Cat and Kitten programming languages were the first major concatenative languages to support Hindley-Milner based static systems [Pur17; Dig18]. We significantly extended their type systems with support for recursive definitions and concurrency. The Joy programming language is one of few concatenative languages with well-defined formal semantics [Thu08]. Earlier efforts extended Forth with Occam-like concurrency primitives, although these were not statically typed [Hen98], and a separate effort introduced linear types to Forth [Bak94]. We believe that Statick is the first programming language to explicitly use a combination of affine, linear, and dependent types for managing communication resources in addition to supporting Hindley-Milner based type deduction.

Chapter 3

The Stannel ISA and Processor

We implemented the Stannel ISA and processor in two stages. We initially wrote an instruction-level software simulation of Stannel along with an assembler and linker, which produced bytecode from a textual representation of Stannel Assembly. We then used the software simulation as a specification for the processor, and implemented it in Verilog.

In the following, a *core* refers to an individual execution unit that executes a single *process* at one time. Processes may create or destroy heap-allocated *channels*, send messages on them, or receive messages. Processes may also spawn new processes. Message communication is a synchronised, blocking operation in the sender and receiver. A *processor* refers is a set of cores, scheduling components, and messaging components. A *System-on-a-Chip* refers to a processor and components capable of receiving programs over USB, executing them, and then returning the result of execution over USB. Our implementation is a complete SoC on the Lattice iCE FPGA. We chose this execution model to admit straightforward testing of programs on the FPGA, but it would also possible to persist programs on the FPGA without USB communication.

Given the constraints of the Lattice iCE FPGA described in Section 2.1 we chose to restrict the clock speed to 16 MHz rather than the maximum clock speed of 100 MHz. Although 100 MHz is the maximum clock speed, it is not possible to reliably perform memory transactions any faster than 50 MHz [Ver18], which initially motivated our reduction in maximum clock speed. Modelling and timing analysis of the design suggests the maximum clock speed is 34.1 MHz, but we found that the tools could not reliably synthesize the design at this clock speed. We therefore chose to reduce the maximum clock speed to 16 MHz as this matches many other microcontrollers we hoped to compete with, including the BBC micro:bit [Mic18a].[1] We configured process memory cells with 8-bit address and 16-bit words.

A core can therefore perform one memory transaction per memory cell in our design. The Lattice iCE FPGA has 32 memory cells, and it is possible to interact with each of these independently. To execute an instruction every cycle the core must separately fetch instructions whilst (possibly) reading or writing data. Therefore our design uses the Harvard architecture: some memory cells store program instructions whilst others store program data.

[1]Last year we developed a JIT compiler for the BBC micro:bit, a microcontroller with a 16 MHz Nordic Semiconductor Cortex-M0 core, which draws 250 µA MHz^{-1} (4 mA total) [Nor14], whilst the Lattice iCE HX8K draws 1.14 mA [Lat17].

Unlike traditional architectures, our design performs all allocation, scheduling, and messaging operations in hardware. This significantly reduces latency for these operations, with most communication operations occurring in fewer than 20 cycles.

3.1 Processor

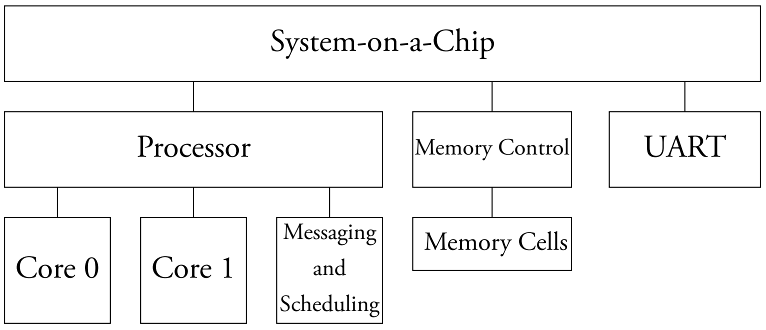


Figure 3.1: Block diagram for the Stannel architecture

Our implementation separates cores from memory to support virtual memory. The controller associates each process or controller component with a memory cell. A core running a process p can only directly write to the cell associated with p. The memory controller supports M components accessing N memory cells simultaneously and independently — its behaviour is undefined if multiple components attempt to access the same memory cell.[2]

Our implementation supports 2 cores accessing 16 process memory cells, 2 instruction cells, and 4 memory cells for storing scheduling and communication information. These restrictions are due to the limited number of logic units, memory cells, and wiring capacity of the Lattice iCE FPGA. Our design utilised approximately 93% of the logic units on the design, and we estimate that our memory controller consumed around half of this footprint.[3] However, our design is parameterised such that any of these constraints can change for a larger or more powerful FPGA.[4]

The UART module communicates over USB with the host computer to receive programs and send the resulting state of the processor. When it receives programs it concurrently writes them to two memory cells, which can later be accessed by each core. When the UART module fully receives a program it transfers control to the processor, which schedules a process beginning at the first instruction one of the cores (using the scheduling component). The program then executes. Once the scheduling component determines that no further processes can execute — which occurs once all processes terminate in a deadlock-free program — control returns to the UART component, which then outputs the contents of each memory cell. On the host computer a script then executes, which verifies that the contents of memory (a) match the simulation and (b) match the expectation of the user, which is provided with the original program code.

[2]Rather than directly implementing the memory controller in Verilog we wrote a script that generates it.

[3]YoSys, the suite of tools we used, does not report per-module resource usage. We determined this figure by disabling parts of the design and rebuilding.

[4]The default word and address sizes can also be changed, although our implementation assumes that all instructions are represented in exactly one byte.

3.2 Instruction Set

Rogers' Stack Virtual Machine [Rog17] and the original Inmos Transputer [Inm87] are our primary inspirations for the instruction set, and they (generally) encode an instruction in 1 byte, and our ISA follows this decision. This is notably smaller than the 2 byte instructions used by ARM Thumb, or variable length Intel x86 instructions. Our prior work found 1 byte instructions and a stack-machine architecture admit more compact code than larger instructions for a register-based architecture [Den18]. A program is a sequence of 1 byte instructions with the initial process beginning at instruction 0.

Unlike the original Transputer architecture, which used a set of 3 registers as a stack and a separate in-memory value stack, our design uses a single value stack. We made this choice to simplify the instruction set (as there is no need for instructions that transfer values between registers and the main stack). In order to reduce the latency of key operations we maintain a 'cache' of the top 3 elements of the stack in registers. Values are read and written from these registers as needed.

To simplify encoding and decoding, each instruction is divided into an *opcode* and *operand*; in most cases the operand is an extension of the opcode. We have distinct opcodes for arithmetic operations, stack operations, jump operations, function operations, and process/messaging operations; Appendix A details the binary encoding. Our ISA also includes dedicated instructions for pushing integer constants that cannot be encoded in the four-bit operand.

We follow the Transputer's approach to process initiation and termination [Inm85]. Any process can start a new process, but a process can only halt or deschedule itself. The 'start process' pops a start address from the stack, and then moves an arbitrary number of words to the initial stack of the new process.

We diverge from the Transputer in our approach to channels. In Occam, the programming language of the Transputer, a fixed number of channels were created statically [Hyd95]. We take an approach inspired by more recent languages featuring channels, such as Go, and instead model channels as heap-allocated resources, which are created dynamically during program execution. We restrict channels to communicating a single word of data to simplify heap allocation, as decribed in Section 3.4.1. In addition to instructions for creating and destroying channels and performing synchronised send (`!`) and receive (`?`) operations, we also provide instruction-level support for alternations, allowing a process to wait on several channels at once, and resume after receiving a message on any one of them.

3.3 Core

Each core is either inactive or executing a single process. Each core retrieves instructions from separate, dedicated memory cells. A core can also interact with an arbitrary memory cell that stores the call stack and value stack.

Each core has registers for the program counter, call stack pointer, stack pointer, and top 3 elements on the stack.[5] When the core executes an instruction that could lead to the descheduling of the current process it will write these values to memory so that the process can be later resumed.

[5]The ALU also stores condition flags; see Appendix A.2.

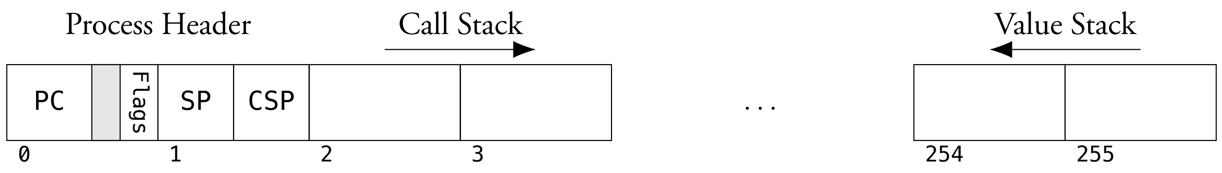


Figure 3.2: Processes store register state in a header. The first 9 bits are the program counter (one of 512 addresses), and 4 bits of the first word are also used for storing flags — leaving 3 bits unused. The stack pointer and call stack pointer address are in the subsequent word, and the remainder of the cell is used for the call stack, which grows upwards, and the value stack, which grows downwards. For simplicity, program counters on the call stack are zero-extended and stored as 16-bit words.

Our design adopts a 2-stage pipeline to parallelise instruction fetching and execution. The fetch stage retrieves the next instruction to execute, and then passes the instruction onto the execution stage on the next cycle. The execution stage performs one or more of the following operations:

- Mutate the values in registers representing the top three elements of the stack;
- A memory transaction that affects the call stack or value stack;
- A second memory transaction that will stall the fetch pipeline for a single cycle;
- A jump (or call) that changes the next program counter, which will stall the execution unit for a cycle whilst the fetch unit retrieves the new instruction; or
- Perform a communication or scheduling operation.

The core is modularised around the pipeline stages. The aforementioned registers are stored and manipulated exclusively in the execution module, and they can be updated when an instruction needed to perform a secondary I/O operation during its execution. Additional modules support reading and writing register values to a process's memory cell.

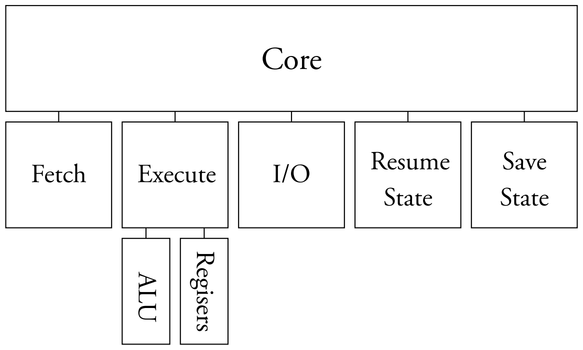


Figure 3.3: The modular layout of a single core; each core is identical.

3.4 Scheduling and Messaging

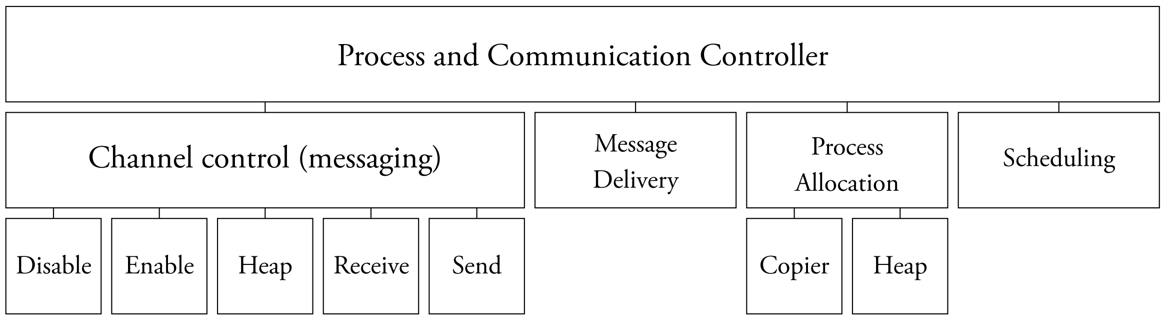


Figure 3.4: Modular layout of the processor controller.

When an individual core encounters a communication instruction or an instruction that affects process state, it issues a signal to the processor's controller component (see Figure 3.1). The controller handles a message from a single core at once, and prioritises them in round-robin order. After receiving a message from a core, the controller determines which process sent the message, and then performs the operation:

- If the operation involved communication over channels, a module determines if a process should be scheduled, if a process should be descheduled, and which process to deliver the message to;
- If a message needs to be delivered to a process then it's either written to memory (if the process is not actively running on a core) or delivered directly to an active core;
- If a process needs to be allocated due to a process start instruction then a new process is allocated, scheduled, and data is moved from the stack of the initiating process to the new process using a copying module;
- If processes need to be scheduled, then the scheduler's state is updated. Cores may then switch process or become inactive.

Each module is a state machine, and the overall module is a hierarchal state machine that delegates control to sub-components during their execution. It would be possible to pipeline this state machine to support messages from, but we determined that this was beyond the scope of our project.

3.4.1 Channel Communication

In Stannel a channel is a pointer to a heap allocated resource. When an instruction requests the creation of a new channel, an allocator allocates a new address for storing data related to the channel. We restrict channels to communicating a single word at once to simplify wiring and reduce the complexity of the heap allocator: if all allocations are the same size then heap fragmentation is not an issue.[6] The heap allocator is a straightforward state machine: it will initially fill a single memory cell by allocating at the end of the heap and then allocate using a linked list of freed addresses. Allocations at address 0 never occur, so it denotes a 'null pointer'.

[6]In practice these restrictions are reasonable. Dedicated processes can model buffered channels whilst alternations can model many-to-many channels.

Each channel stores two words: the first is the ID of a process and the second is the message. Initially the process ID is 0, denoting that no process is actively using the channel. When a process sends a message, the controller will check whether a process is waiting on the channel, and if not then write its ID (along with the message) to memory and deschedule the process. Alternatively, if a process ID has been written then it will write the message to memory, schedule the receiving process, and continue execution. Whichever process checks the process last will reset the process ID to 0. This approach is safe as only a single core's send or receive request is handled at once.

Support for alternations is more complex, and we take our approach from the original Transputer architecture [Inm85; Inm88]:

1. When a process enters an alternation it is added to a set of processes in alternations, a bit-field in hardware.
2. The channel in each of arm of the alternation is then 'enabled' where the process ID is written to the memory associated with the channel (and marked with an extra bit to indicate it is in an alternation). If a process has already sent a message to the channel then the receiving process is remove from the alternation set.
3. Once the channels are enabled the process is descheduled until a message is sent to one of its channels, at which point it is resumed.
4. The process 'disables' the channel of each arm of the alternation. When it encounters the first channel to have received a value it will determine the destination instruction to jump to.
5. After disabling all the channels the process explicitly leaves the alternation and resumes execution on the correct arm of the alternation.

For simplicity, our design assumes that channels in an alternation are ordered by priority, rather than non-deterministically selecting the arm in the event of multiple channels sending a message.

The architecture does not perform any validation on sending/receiving process IDs, channels in alternations, or freed channels; communication is a low-level operation. To ensure that communication is safe and such errors are not possible, we introduce the Statick programming language in Chapter 4.

Our architecture is extensible, so that on a larger FPGA it is possible to increase the maximum number of processes or cores. Furthermore, the interface for a 'core' is generic: its only requirement is that it can receive and send messages to the processor's controller for channel operations. To support features such as secondary modules, e.g. for optimised machine learning workflows, or for communication over 'hardware' channels, extra non-compute cores that communicate over fixed channels could be introduced: new hardware will not require new instructions.

3.4.2 Process Allocation

When a new process is initiated another heap allocation module determines which cell to use.[7] After the process's memory cell is allocated a secondary module copies an arbitrary (provided) number of words from the creating process's stack to the new process's stack. The newly created process is then rescheduled.

3.4.3 Scheduling

The scheduler determines which processes are active on which cores and which processes to schedule next. For simplicity, we implemented a round-robin scheduler: when a new process is scheduled it is appended to the end of a 'schedule queue', and when a process is descheduled, the next process is popped from the front of the queue. Typical scheduling algorithms in processors and operating systems have a notion of a 'quantum', i.e. a short period of time that a process can execute for before the process is temporarily descheduled so that other processes can compete for resources [BO16]. We chose not to include this feature in our scheduler as other embedded systems generally do not include it so that sequential program execution time is deterministic.

[7]This heap allocator uses a separate memory cell to the channel allocator for simplicity, but there is no technical reason for them not to share the same cell — the heap allocator is structured so that multiple allocators could use different regions of the same cell, although not concurrently.

Chapter 4

The Statick Programming Language

Stannel, as presented in Chapter 3, supports writing programs in an assembly language, but this is not an effective way to write large programs. Writing assembly is a tedious exercise at best, and foolish at worst — the Stannel assembler performs almost no static verification of programs. It is therefore desirable to instead program in a language that supports all aspects of the instruction set, but has higher-level control flow primitives and a static type system to avoid common errors.

Concatenative languages support stack-based programming. If a value on the stack has a resource associated with it, e.g. a pointer to memory on a heap, then the resource can be freed when the stack value is dropped. Many programming languages including Kitten[1] and C++ utilise this notion, but it isn't sufficient for shared resources: if each owner frees a resource after it finishes using it then the heap would become corrupted. Existing languages mitigate the need for the programmer to manually manage memory with run-time techniques such as garbage collection and reference counting or compile-time techniques such as borrow checking.

Channels are heap allocated resources in Stannel, so they must be heap allocated in Statick too. Channels, as described in Chapter 3, are also shared resources; a receiving process and a sending process store a pointer to a memory location that they can use to read and write messages. Freeing a channel when either its sender or receiver is dropped from the stack is a good first step towards automatic memory management, but it does not prevent the other user of a channel from attempting to use the channel after it has been freed (unless, at runtime, some other precautionary step is taken). If it is also possible to duplicate channel pointers then it remains possible to double-free a pointer to a channel. We demonstrate prevention of these bugs at compile time using a static type system. In Statick, there are two kinds of channel, each with different guarantees:

- **Use *forever* channels:** once created, these channels guarantee that it is always possible to send and receive a value on them, i.e. a sender or receiver waiting on the channel will always eventually be awoken
- **Use n channels:** once created, these channels guarantee that exactly n values will be sent on them, after which they can be freed

[1]A statically typed concatenative language [Pur17].

The static type system of Statick only prevents errors in the memory management of channels, rather than their general mismanagement. For example, it remains entirely possible for two processes to send and receive messages on channels in a different order, leading to a deadlock. Modelling a Statick program with process calculi such as CSP would detect these errors.

Figures 4.1 and 4.2 presents a sample Statick program where a sender communicates exactly one value to a receiver. Figure 4.2. All functions take a (generic) stack as input and output a (generic) stack. When the channel is created, its type describes that we expect exactly one value to be communicated on it, and the type system enforces that the channel cannot be destroyed until that occurs. Each communication operation decreases the number of communication operations remaining on the channel by 1 and the channel can be destroyed (through the `del` standard library function) when it has no uses remaining. It is only possible to destroy the 'receiver' (Rx) variant of a channel and not the 'transmitter' (Tx) variant: this ensures that a channel is freed exactly once, and only after it's no longer needed by sender or receiver.

```
main =
  chan$_1$          -- Creates a pair (receiver, transmitter) for a channel
                 -- that may be used once
  'sender proc$_1$ -- Creates a new process from the function 'sender' by
                 -- moving one word from this process's stack
  ?              -- Receive a value on the channel
  swap del       -- Swap the value with the channel and frees the channel
sender =
  42 ! drop      -- Sends the value on the channel and drops the used
                 -- (sending) channel without freeing
```

Figure 4.1: A Statick program that sends a single value on a channel from one process to another.

```
main =       :: $S \to S$
    chan$_1$     :: $S \to S \times \mathsf{chan}(1, \mathsf{Rx}, \alpha) \times \mathsf{chan}(1, \mathsf{Tx}, \alpha)$
    'sender  :: $S \to S \times (S' \times \mathsf{chan}(1, \mathsf{Tx}, \mathsf{int}) \to S')$
    proc$_1$     :: $S \times \alpha \times (S' \times \alpha \to S'' : \mathtt{NoConsumeableOrUndroppableTypes}) \to S$
    ?        :: $S \times \mathsf{chan}(n+1, \mathsf{Rx}, \alpha) \to S \times \mathsf{chan}(n, \mathsf{Rx}, \alpha) \times \alpha$
    swap     :: $S \times \alpha \times \beta \to S \times \beta \times \alpha$
    del      :: $S \times \mathsf{chan}(0, \mathsf{Rx}, \alpha) \to S$

sender = :: $S \times \mathsf{chan}(1, \mathsf{Tx}, \mathsf{int}) \to S$
    42   :: $S \to S \times \mathsf{int}$
    !    :: $S \times \mathsf{chan}(n+1, \mathsf{Tx}, \alpha) \times \alpha \to S \times \mathsf{chan}(n, \mathsf{Tx}, \alpha)$
    drop :: $S \times \alpha : \mathtt{Droppable} \to S$
```

Figure 4.2: The program of Figure 4.1 annotated with the type of each function.

Statick supports alternations for listening on multiple channels, and the type system ensures that the type of the stack is the same after executing *any* arm of the alternation. Further examples are presented in Appendix D.

4.1 Grammar

Statick programs are a list of *definitions*, which associate a unique identifier with a term. A *term* is a list of expressions execute from left-to-right. An *expression* is either a language primitive or the name of a definition. Each Statick program includes the Statick Standard Library. The abstract grammar of Statick is presented below.[2] A sequence of 1 or more elements is denoted by an ellipsis.

$$\begin{aligned}
\text{Program} &::= \text{Definition}\ldots \\
\text{Definition} &::= \text{Identifier} \mapsto \text{Term} \\
\text{Term} &::= () \mid \text{Term} \cdot \text{Expression} \\
\text{Expression} &::= \text{Name} \mid \mathsf{Ref}(\text{Name}) \mid \mathsf{Anonymous}(\text{Term}) \mid \mathsf{If}(\text{Condition}, \text{Term}, \text{Term}) \\
&\quad \mid \mathsf{While}(\text{Condition}, \text{Term}) \mid \mathsf{Repeat}_k(\text{Term}) \mid \mathsf{Alternation}(\text{Arm}\ldots) \\
\text{Name} &::= \text{Identifier} \mid \text{Identifier}_{n \in \mathbb{N}} \\
\text{Condition} &::= \text{Term} \\
\text{Arm} &::= n \in \mathbb{N} \to \text{Term} \\
k &::= n \in \mathbb{N} \mid \infty
\end{aligned}$$

Natural numbers, booleans, and other values are absent from Statick's grammar. Instead, these are functions in the standard library that place a value at the top of the stack. We could have expressed all langauge constructs as functions in the standard library too, but because they affect control flow we instead promoted them directly into the grammar of the language. Furthermore, as language constructs their presence is clearer in source code; we believe **`if`** `(cond)` **`then`** `(cond_true)` **`else`** `(cond_false)` is more familiar (and more efficient to compile) than `(cond) (cond_true) (cond_false)` **`if`**.

A natural number parameterises alternation arms, which represents the offset on the stack of a channel to listen on. An alternative here would be a term that, once evaluated, returns a channel. However, it then becomes harder to (statically) enforce that a process isn't listening on a channel more than once in the same alternation, but this is straightforward with static offsets. Natural numbers can also parameterise names from the standard library and the Repeat construct.

4.2 Semantics

We present the semantics of Statick as executing on a theoretical computer. In Chapter 5 we discuss the steps taken to ensure that the compiler generates code that executes programs that follow the specification. We only discuss the execution of a single process, which executes on a single *Statick Virtual Machine*. We assume an environment that supports creating, executing, and destroying multiple Stannel Virtual Machines, potentially in parallel. The communication model of channels and processes follows the model of CSP: we assume the existence of a fair scheduler that synchronises communication operations.[3] The environment also contains a shared global map Σ of names to terms, i.e. Σ is derived from the list of definitions in the program. Σ[`name`] retrieves the term with `name`.

[2]The concrete grammar, and its mapping to the abstract grammar, is presented in Appendix B.

[3]The implementation of Stannel also follows this model, although it only supports a finite number of processes.

There are 4 possible states for a Statick Virtual Machine:

$$\langle k, v\rangle \mid \mathsf{Waiting}(k, v, c, m) \mid \mathsf{Awaiting}(C) \mid \bot$$
$$\text{Operation} ::= \text{Expression} \mid \mathsf{ConditionalBranch}(\text{Term}, \text{Term})$$

k is a sequence of operations. v is a stack of values, where a value is either an integer, boolean, term, a Repeat's counter, or channel.[4] C is a set of triples $\langle c, k, v\rangle$ where c is a channel. By default, a Statick VM begins in the state $\langle\text{main}, []\rangle$, where 'main' is the name of the definition to use as the entry point. It then progresses according to the rules:

$$\begin{array}{rcl}
\langle [], v\rangle & \to & \bot \\
\langle \text{name} : k, v\rangle & \to & \langle \Sigma[\text{name}] +\!\!+\ k, v\rangle \\
\langle \mathsf{Ref}(\text{name}) : k, v\rangle & \to & \langle k, (\Sigma[\text{name}]) : v\rangle \\
\langle \mathsf{Anonymous}(t) : k, v\rangle & \to & \langle k, t : v\rangle \\
\langle \mathsf{If}(c, t, f), v\rangle & \to & \langle c +\!\!+\ \mathsf{ConditionalBranch}(t, f) : k, v\rangle \\
\langle \mathsf{While}(c, b) : k, v\rangle & \to & \langle c : \mathsf{ConditionalBranch}([b, \mathsf{While}(c, b)]) : k, v\rangle \\
\langle \mathsf{Repeat}_n(t) : k, \mathsf{Counter}(n-1) : v\rangle & \to & \langle k, v\rangle \\
\langle \mathsf{Repeat}_n(t) : k, \mathsf{Counter}(n' \neq n) : v\rangle & \to & \langle t : \mathsf{Repeat}_n(t) : k, \mathsf{Counter}(n'+1) : v\rangle \\
\langle \mathsf{Repeat}_n(t) : k, v\rangle & \to & \langle t : \mathsf{Repeat}_n(t) : k, \mathsf{Counter}(0) : v\rangle \\
\langle \mathsf{Repeat}_\infty(t) : k, v\rangle & \to & \langle t : \mathsf{Repeat}_\infty(t) : k, v\rangle \\
\langle \mathsf{Alternation}(A), v\rangle & \to & \mathsf{Awaiting}(\{\langle v[n], t +\!\!+\ k, v\rangle \mid (n \to t) \in A\}) \\
\langle \mathsf{ConditionalBranch}(t, f) : k, \mathsf{true} : v\rangle & \to & \langle t +\!\!+\ k, v\rangle \\
\langle \mathsf{ConditionalBranch}(t, f) : k, \mathsf{false} : v\rangle & \to & \langle f +\!\!+\ k, v\rangle
\end{array}$$

Additionally we present the semantics of a limited number of standard library functions that can't be denoted in Statick alone. We do not include the binary operators $+, -, <, >, \leq, \geq, =, \neq$, `and`, `or`, the unary operator `not`, and the stack operators `drop`, `dup`, `swap`, `rot`, `tuck` in the description below as these correspond directly to Stannel instructons in Appendix 3.

[4] Terms on the stack are always denoted on the stack surrounded by parentheses. This is to make clear the distinction between true — a value — and (true) — a term.

$$\begin{array}{rcl}
\langle \texttt{@}_n : k, v\rangle & \rightarrow & \langle k, v[n] : v\rangle \\
\langle n \in \mathbb{N} : k, v\rangle & \rightarrow & \langle k, n : v\rangle \\
\langle \texttt{true} : k, v\rangle & \rightarrow & \langle k, \text{true} : v\rangle \\
\langle \texttt{false} : k, v\rangle & \rightarrow & \langle k, \text{false} : v\rangle \\
\langle \texttt{apply} : k, t : v\rangle & \rightarrow & \langle t +\!\!+ k, v\rangle \\
\langle \texttt{chan}_n : k, v\rangle & \rightarrow & \langle c : c : v\rangle \\
\langle \texttt{!} : k, c : m : v\rangle & \rightarrow & \mathsf{Waiting}(k, v, c, m) \\
\langle \texttt{?} : k, c : v\rangle & \rightarrow & \mathsf{Awaiting}(\{\langle c, k, v\rangle\}) \\
\langle \texttt{del} : k, c : v\rangle & \rightarrow & \langle k, v\rangle
\end{array}$$

The functions `!` and `?` are optionally parameterised with offsets: $\texttt{!}_n$ corresponds to $\texttt{@}_n$ `!`.[5]

When the system detects a virtual machine entering the Awaiting state it will begin to perform message delivery.

The numeric parameter of `chan` is optional and affects the type of the created channel; the virtual machine makes no distinction between the earlier described 'forever' and 'use n'. This function produces two copies of the same channel value, which are separately used for receiving and transmission (the VM doesn't distinguish them).[6]. Note `chan` and `del` create and destroy shared resources in the environment; an individual Statick VM does not concern itself with the implementation of these functions.

When a process sends a value on a channel c it enters the $\mathsf{Waiting}(k, v)$ state. We presume the environment of Statick VMs will then resume that process in state $\langle k, v\rangle$ when another process receives the message on channel c. Symmetrically, if a process is in state $\mathsf{Awaiting}(C)$ with $\langle c, k, v\rangle \in C$, and another process sent a message m on c then we assume the environment resumes the process in state $\langle k, m : v\rangle$.[7]

[5]We include these functions in the standard library as these terms can't be typed.

[6]In a practical implementation, such as Stannel, a channel 'value' represents a pointer to the data associated with the channel

[7]The type system of Statick *programming language* ensures that two processes cannot be waiting on the same channel at once. In the Statick *Virtual Machine* the same assumption is not made, and instead a process will be non-deterministically selected amongst the processes waiting on c.

4.3 Types

The structure of types and stacks in Statick is presented below:

$S ::=$	A	An element from the set of stack variables **S**
	$\mid S \times T$	A non-empty stack
	$\mid \bot_S$	The stack of programs that do not terminate
$T ::=$	α	An element from the set of type variables **T**
	$\mid S \to S$	Function
	$\mid \mathbb{N}$	Unsigned integers
	$\mid \mathcal{C}$	Counter for finite Repeat
	$\mid \mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$	Booleans
	$\mid \bot_T$	The void type
	$\mid \mathsf{chan}(N, D, T)$	Channels
$N ::=$	x	An element from the set of channel use variables **N**
	$\mid n \in \mathbb{N}$	
	$\mid \infty$	
	$\mid \mathsf{succ}(N)$	
$D ::=$	$\mathsf{Receive} \mid \mathsf{Send}$	

The type system of Statick is based on earlier work on the programming languages Cat and Kitten; the first major concatenative programming languages to feature a static type system [Dig18; Pur17]. Further work implemented their type systems within Haskell's type system [Han12].[8] The Cat type checker does not support recursive definitions, unlike most implementations of type checkers based on the Hindley-Milner algorithm, and as such the type checking algorithm used by the Statick compiler is derived from earlier work [Car87], rather than the Cat type checker.

The sets **S**, **T**, and **N** are disjoint. In the following text uppercase Latin letter range over stack variables,[9] lowercase Greek letters range over type variables, and lowercase Latin letters range over channel use variables.

The number of times that a channel may be used is based on adapted form of Peano arithmetic, with the axiom

$$\mathsf{succ}(\infty) = \infty$$

We abbreviate $\mathsf{succ}(n)$ to $n + 1$ in the remainder of the text.

In our compiler we restrict $\mathbb{N}$ to $[0, 2^{16})$ in all cases, i.e. the integers represented by 16-bit unsigned integers, which are natively supported by Stannel.

[8] It's not possible to model the Statick type system using the Haskell type system at the time of writing as it doesn't support dependent types.

[9] Generally the letter S is used, but note that this is distinct from the grammatical definition of all stacks.

Types also have an associated set of constraints, such that a type variable with a set of constraints C can only be replaced by a type that satisfies C. Stacks can also be similarly constrained. These constraint systems are used to enforce safe channel operations.

4.4 Type Checker

Our type checking algorithm is based on the traditional Hindley-Milner type checking algorithm, as presented in [Car87], with additions from earlier work to support unification of stack variables with stacks of variable arity [Kut02]. Our type checking algorithm has two operations: typing expressions and unifying types or stacks. We present the rules for typing expressions in Appendix C and the unification algorithm below:

1. An initial environment mapping names to types, Γ, is constructed. Each declaration is initially added with the generic type $\forall\, S, S' \,.\, S \rightarrow S'$.[10]

2. An implicit program graph $\mathcal{G}$ is constructed where each vertex denotes a name and each directed edge denotes a dependency from one function to another (i.e. there is an edge $a \rightarrow b$ if there a calls or refers to b). A topological sort of $\mathcal{G}$ is found.

3. Names are iterated in topological order and annotated with the type of the corresponding term, per the rules described in Appendix C.

4. A second traversal deals with recursive definitions. We attempt unification between the actual type of a name and the inferred type of a name (whether in a call or reference). If unification fails then the function was called with the incorrect arguments.

The final unification step disallows writing Statick programs with unbounded recursion. A function may make a recursive call to itself so long as there is a branch of the function that returns without making a recursive call. In cases where a function never terminates (due to a Repeat_∞ loop) the return stack $\bot_S$ is inferred instead.

A unifier is a function that maps elements from **S**, **T**, and **N** to stacks, types, and channel uses respectively. We denote the replacement of x by X in each of these contexts as $[X/x]$ and $[X/x] \circ U$ denotes performing the replacement of x by X after performing all steps of replacement in U. The empty unifier is $\mathbb{E}$.

The construction of a most general unifier is based on Robinson's Unification Algorithm:

1. Begin with types A and B.[11] Let $U = \mathbb{E}$

2. Compute $U(A)$ and $U(B)$; terminate if the types are equal

3. If the types are not equal then they are of the form

[10]Most standard library functions are also added, although some of them are lazily created.
[11]The algorithm naturally extends to stacks so we omit its description here.

$$U(A) = t_1\, c\, t_2$$
$$U(B) = t_1\, C\, t_3$$

where $c \neq C$. If it is possible to unify them, then we let $U = [C/c] \circ U$ and return to step 2, otherwise terminate with an error.

- If C is a type and c is a type variable then the types are unifiable if c does not occur in C and if c has any constraints associated with it then C satisfies those constraints:
  - **Droppable:** All types can be dropped from the stack, except for channels and counters. Transmission channels with a finite number of uses remaining can be dropped from the stack if we can unify that number of uses with zero; receiving channels may alternatively be dropped with the `del` standard library function, which also frees their resources. Infinite-use channels can never be dropped. By ensuring that the `drop` function is the only function that can remove elements from the stack (without transforming their type in some way) we ensure that it is the only function equivalent to the *weakening* operation of typed λ-calculus.
  - **Duplicable:** All types except channels and counters may be duplicated.
  - **Integer like:** Allows a type to be copied and converted to an integer. Integers, booleans, and counters satisfy this property.
  - **Must consume:** Requires that the type is used at some point in the evaluation of a function, i.e. that if it appears on the left of a function then it doesn't appear on the right of the function.
- If C is a stack and c is a stack variable then the types are unifiable if c does not occur in C and C satisfies constraints associated with C:
  - **Does not contain must consume types or types that cannot be dropped:** This constraint is used to prevent 'forgetting' that a stack contains a type that ought to be used; the stack variable at the bottom of the stack that each process starts with must have this property, and the entire terminal stack of a process must have this process.
  - **Must be base:** Only true for stack variables at the bottom of stack.
- If C is a channel use property and c is a channel use property variable then unification succeeds if C is infinite, C doesn't contain c, or C does contain c, in which case we replace with ∞. This final rule means that if a channel is used inside an infinite loop then we can infer that it is an infinite use channel.

The following properties hold:

- Non-exhausted channels are not 'droppable' so cannot be destroyed;
- The most generic type for each function is inferred;

- If an expression's generic type includes a channel on its left hand side then that function uses the channel some way (this follows from the previous property); and
- It is not possible to manipulate the counter of a loop with any operations or functions other than the Repeat construct.

Theorem 1. A 'use N' channel must have N values sent or received on it.

Proof. We require that a process does not terminate with a channel left on its stack. Once created, a channel either remains on the stack until process termination, or it is used by a function later in the process's execution.[12]. The sending and receiving operations decrease the number of uses a channel still has by 1, and it's only possible to destroy a channel once it has been used N times. Therefore after a 'use N' channel is created we guarantee that N values are *eventually* sent on it.

□

Theorem 2. A 'use forever' channel must have an infinite number of values sent or received on it.

Proof. A process cannot terminate with a 'use forever' channel on its stack. However, when the channel operations use such a channel they do not decrease the number of uses remaining on the channel, so instead the process must terminate by using the channel in a infinitely repeating loop.

□

4.5 Compiler

The Statick compiler was implemented as a Rust program. The concrete syntax of the language is described in Appendix B, and a recursive descent parser implements this. The program is then type checked by an implementation of the algorithm described in Section 4.4.

There is no syntax in the concrete grammar for denoting a function or expression's type. We made this decision firstly to expose the Hindley-Milner-derived type system of Statick and secondly to simplify the parser. However, we appreciate that a programmer may like access to their program's types, so the Statick compiler can output each definition's type, in line with tools such as GHCI.

After type checking completes the compiler then performs code generation. Most standard library functions translate to a small number of Statick instructions. To produce faster code the compiler then unifies neighbouring labels, so `L0: L1: ...` simplifies to `L0: ...`. Labels followed by a jump unify to the jump location (if it can be statically determined). A label is considered unused if it is not directly jumped to. If the preceding block ends with a branching instruction (i.e. a return, function call, conditional jump, or unconditional jump) then all code after an unused label but before a used label is removed. Otherwise, if the label is not directly referenced and the preceding code flows directly into it then the label is removed. This ensures that most 'dead' code is eliminated from the final program. In compiler literature, a basic block is a sequence of instructions that are always entered at the same point,

[12]Possibly by passing it to another process, which must use it in some way.

executed in the same order, and terminated by a branch instruction [App98]. Code between labels now forms basic blocks, so a peephole optimiser is applied to the basic blocks of the program.[13]

The language is constructed so that only a single implementation of each definition must be compiled, even though definitions can be polymorphic, in order to reduce the output code size and simplify the code generator. Consequently, channels must be manually managed through the `delete` function, rather than just dropped or forgotten. The type system is instead constructed to enforce the application of `delete` so that a valid program cannot be constructed that doesn't free channels once they have been used. In Section 6.2 we consider extensions of the language that would monomorphise multiple versions of generic functions so that explicit management of channels is no longer required.

Finally, an assembler converts the Statick Assembly into bytecode that can be directly executed on the processor.

[13]The peephole optimiser is deliberately small, as the primary focus during development was the type checker. Furthermore, we produced an optimising JIT compiler for stack machines that executed on micro-controllers [Den18], and believe that the two projects could be combined.

Chapter 5

Testing

5.1 Stannel

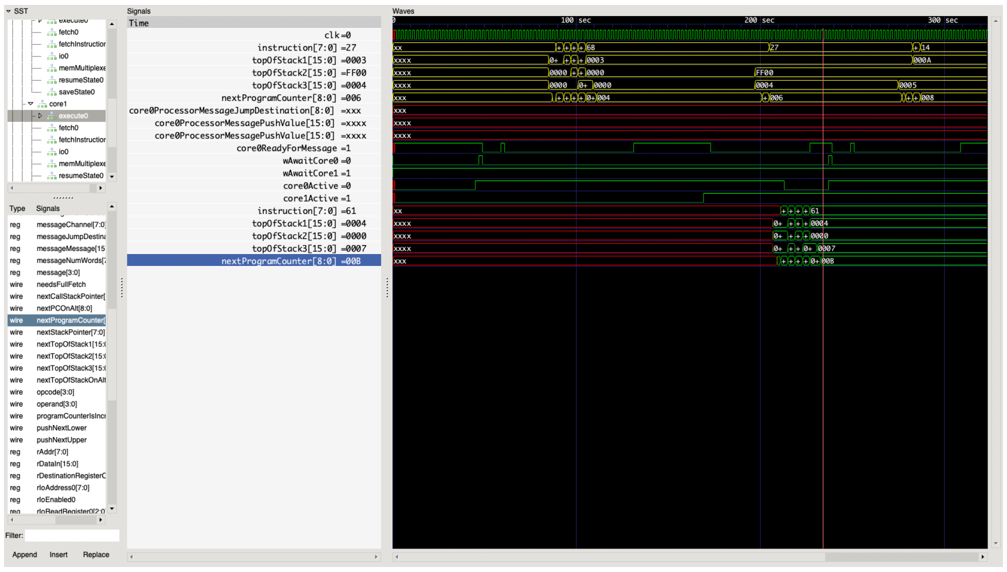
As outlined in Chapter 3, we designed an instruction-level simulator of Stannel before implementing the actual hardware. We tested each module within the simulator using unit tests, and secondarily tested its execution of over 100 test programs. These tests were then adapted to run on the hardware once implementation was complete.

5.1.1 Correctness of Components

Our approach does not formally prove the correctness of each component, nor that the components communicate correctly. Formal verification tools (using SAT solvers) are available for Verilog. We determined at the outset that time constraints would preclude us from fully formally proving the correctness of our design. Generally these tools prove that a property always holds for the first n cycles of execution or that a property always holds by induction. All our modules are state machines that reach a terminal state within a fixed, finite number of states, which would therefore motivate the first approach. However, either approach requires the programmer to restrict the proof assistant's state space so that the tools do not explore states that are impossible to reach [Gis18]. After assessing whether or not formal verification was appropriate we determined that either approach would require such significant configuration of the state space that a simpler, more informal test-based approach was adequate.

Our design contains 22 Verilog modules, with many modules re-used throughout the design. Each component was separately tested with a 'test bench', a separate Verilog module that, in conjunction with a Verilog simulator, initialises the component and tests it under different inputs, validating that it then produces the correct output in a deterministic number of clock cycles. In total, we wrote 156 tests, which fell into one of three categories:

- **Tests validating the component produced the correct output given certain inputs.**
- **Tests validating the component did not alter state unrelated to its input.** For example, when a component interacts with memory it should write to memory that it ought to directly affect, and



```
Time
                              clk =0
                instruction[7:0] =27
               topOfStack1[15:0] =0003
               topOfStack2[15:0] =FF00
               topOfStack3[15:0] =0004
          nextProgramCounter[8:0] =006
core0ProcessorMessageJumpDestination[8:0] =xxx
   core0ProcessorMessagePushValue[15:0] =xxxx
   core0ProcessorMessagePushValue[15:0] =xxxx
            core0ReadyForMessage =1
                     wAwaitCore0 =0
                     wAwaitCore1 =1
                     core0Active =0
                     core1Active =1
                instruction[7:0] =61
               topOfStack1[15:0] =0004
               topOfStack2[15:0] =0000
               topOfStack3[15:0] =0007
          nextProgramCounter[8:0] =00B
```

```
Type  Signals
reg   messageChannel[7:0]
reg   messageJumpDestina
reg   messageMessage[15
reg   messageNumWords[7
reg   message[3:0]
wire  needsFullFetch
wire  nextCallStackPointer[
wire  nextPCOnAlt[8:0]
wire  nextProgramCounter[
wire  nextStackPointer[7:0]
wire  nextTopOfStack1[15:
wire  nextTopOfStack2[15:
wire  nextTopOfStack3[15:
wire  nextTopOfStackOnAlt
wire  opcode[3:0]
wire  operand[3:0]
wire  programCounterIsIncr
wire  pushNextLower
wire  pushNextUpper
reg   rAddr[7:0]
reg   rDataIn[15:0]
reg   rDestinationRegisterC
reg   rIoAddress0[7:0]
reg   rIoEnabled0
reg   rIoReadRegister0[2:0
```

Figure 5.1: GTKWave viewing a waveform for the execution of a simple program on two cores.

all other memory state should remain the same.

- **Validate that components can 'pause' and 'resume' correctly.** A core, for example, needs to stop execution whilst it waits to receive a message from the processor's controller. Whilst in the paused state, its sub-components should not continue executing.

Each component only required a small number of tests because of a small domain for acceptable inputs and deterministic termination. Furthermore, many components are state machines that do not branch; they may just perform a sequence of memory operations. Our approach did not formally verify that execution in *every* state was correct, but we were able to identify that the components execute correctly in every *representative* state. Many tests also implicitly tested several components at once, as our full design is constructed as a hierarchy of state machines. Regardless, these tests checked the behaviour of every Stannel instruction.

We executed tests in Icarus Verilog, a simulator [Ica18], which produces 'wave files' that record the value of every register and wire at each clock tick. These files were viewed in a wave viewer as pictured in Figure 5.1 [GTK19]. These tools identified at what point errors occurred, but manually checking the values of registers is a tedious exercise, especially in a design with hundreds of wires and registers. Therefore our tests were automated with a separate series of macros and scripts to check the state values.

We also developed a script that, in conjunction with a naming scheme strictly adopted in the source code, could detect 'obvious' errors that lead to combinatorial loops.

5.1.2 Execution Correctness

We began testing the processor's execution by writing small programs directly in Stannel Assembly. These programs were initially short sequences of instructions for assessing stack operations, arithmetic operations, branching, and communication. After the development of the Statick compiler it became substantially easier to write test programs in a high-level language, so we developed over 100 test programs.

The correctness of each component was only assessed in a simulation, rather than on the synthesized hardware on the FPGA. To ensure that the processor worked correctly in hardware and in software we developed a separate test suite that:

1. Ran test programs against the instruction-level simulator and collected the final state of the stack of each process;
2. Ran the test programs against the Verilog simulation and validated that the final state of the stack matched the instruction-level simulator; and
3. Transmitted the programs to the FPGA, received the state of each memory cell, and checked that the state matched the simulation.

The first two suites validated the correct execution behaviour of the design, whilst the last two suites validated the synthesis. In general, synthesis and simulation tools should always adhere to the Verilog standard and produce the same output, but faults within the synthesis tools often lead to timing errors or combinatorial loops in the final design, causing divergence that we later rectified.

5.2 Statick

5.2.1 Lexing and Parsing

We invested little effort in testing the lexing and parsing components because errors that occurred in these modules surfaced in the tests of later stages of the compiler. Each stage of the compiler produced distinct errors, and these errors are annotated with the origin of the error in the source code, so in the rare occurrences that parser errors emerged in later stages we could easily identify them. Nevertheless we developed around 15 tests that checked each language keyword and construct generated the correct token or AST node respectively.

5.2.2 Type Checking

We wrote over 50 tests against our type checking algorithm. Our tests form an *informal* inductive proof that if a language construct is typable then it is given a type, and that its type is its most generic possible type. However, the majority of the tests check that the type checker fails to type syntactically correct but untypable programs. These tests check that the type checker produces an error that corresponds to the correct point of failure.

We considered formally proving the correctness of the typing checking algorithm and implementing property-based testing using a derivative of the QuickCheck library, but time constraints prevented us from completing either exercise [CH00; Gal14].

5.2.3 Code Generation and Execution

Our code generation tests compile and execute code using the instruction-level simulator for Stannel, and outputs the assembly so that the same programs are tested in the hardware simulator and on the FPGA using the suite described in Section 5.1.2. Importantly, these tests cover every possible instruction that the code generator can produce, along with every language construct in Statick.

5.2.4 Optimisation and Performance

Optimality of compiled code is not a focus of this project, but our compiler includes a straightforward peephole optimiser, described in Section 4.5. Statick is a lightweight abstraction over Stannel Assembly — its main contribution is type safety — and most standard library functions map directly to 1 or 2 Stannel instructions. We attempt no optimisations that could change the stack's type, because this would affect the execution of other instructions. The correctness tests validate that the optimiser's operations are sound.

Table 5.1 presents the effect of the peephole optimiser in terms of the number of cycles executed required to execute a number of test programs.[1] It's not surprising that the peephole optimiser has little impact on performance as as it contains fewer than 10 rules; the greater performance improvement occurs as the result of collapsing jumps and labels where possible.

	Without optimisations		With optimisations	
Program	Compiled size /bytes	Cycles	Compiled size /bytes	Cycles
Empty	1	33	1	33
Single addition	4	36	2	32
Empty loop	16	49	14	47
Nested loops	26	1565	21	1335
4th Fibonacci number recursively	30	246	30	246
Create empty secondary process	6	66	6	66
Repeat with 3 communications	36	202	36	194
Single communication	16	115	16	115

Table 5.1: Bytecode size and cycle counts for programs with and without optimisation

[1] We executed the programs in the Verilog simulator of the Stannel processor. All numbers include cycles used for scheduling and communication. As processing and scheduling times are unaffected by the peephole optimiser, the absolute reduction in cycle counts is more useful to consider than the percentage reduction.

Chapter 6

Conclusion

In this project we developed Stannel, a new processor architecture, and Statick, a new programming language. Together these motivate an alternative approach for software development in the embedded environment.

Stannel utilises a compact instruction set that can efficiently encode instructions whilst still maintaining performance comparable to similar embedded processors. Its 'channel-first' approach in hardware allows low-latency communication between processes without the need for complex memory hierarchies of other architectures. We independently tested the components of our architecture to verify that they each function correctly, and then verified that the processor is able to execute a wide variety of programs compiled for its ISA.

Statick is a novel concatenative language, and the first of its kind to feature statically-typed channels, and affine, linear, and dependent types to restrict resource usage. Furthermore, we evidenced the correctness of its type checker and compiler through a large range of test programs.

6.1 Reflection

This project was undoubtedly the largest piece of work I completed over the course of my degree,[1] and certainly the one that I'm proudest of. Several years ago a senior researcher at Microsoft Research commented to me that he believed that "by the end of a computer science degree you should be able to construct a computer from scratch." This project was therefore a satisfying conclusion to my degree.

I began by prototyping my processor with a software simulator, which I wrote in Rust, a language I hadn't used before this project. After completing the software simulator I spent around 3 months working on the processor implementation itself. I hadn't used Verilog before I started the project, so my progress was initially slow until I formalised my approach for implementing state machines. I used YoSys, an open-source tool created by reverse engineering Lattice iCE FPGAs. Although the tool is an impressive effort by the open-source community around it, I frequently found it a hindrance. Early in the project I introduced combinatorial loops into my design, but the tool synthesized them happily, so

[1]My implementation of Stannel and Statick total around 20,000 lines of code.

I didn't detect them — I discovered around mid-way through that I could use an alternative tool which would discover and reject them, but it didn't find all 'obvious' errors in my design. I wrote nearly all of my tests concurrently with the implementation.

I wrote the compiler for Statick in less than a month, although I planned the language much earlier — I first started thinking about the notion of resource management for channels over 2 years ago. Implementing the type checker was the most enjoyable part of the project, and I thoroughly enjoyed learning Rust to do it.

6.2 Future Work

The 'channel-first' that Stannel takes to communication opens the possibility of introducing new hardware without the need to introduce new instructions into the ISA. Instead, when the processor needs to perform a new operation it can instead introduce a new component that communicates with the processor using the same protocol as existing channels. A straightforward introduction here would be communication over existing pins and buses available on the Lattice iCE FPGA to allow communication between processors or over USB. FPGAs are also used to prototype new dedicated machine-learning hardware [Jou+17] so new components could be added to the design.

Architectural extensions of Stannel could also adapt the architecture to a desktop computing, where assumptions around fast memory are no longer present, and hardware operations such as register renaming, out-of-order execution, and branch prediction are common. Existing work adapts traditional register renaming schemes to stack machines in hardware [Qia+07], and Stannel could support this.

The Statick language, type system, and compiler offer fertile ground for extension. The language should next support user-definable types and support for monomorphisation in the compiler. Such extensions are possible within the context of Stannel, but the compiler could emit code for other architectures using a library such as LLVM or 'transpiling' the code to Go, another programming language featuring channels.

The formal semantics of Statick can be mapped to CSP. An implementation of this mapping in the compiler could allow it to verify properties of Statick programs such as liveness and deadlock freedom as a compilation stage.

Appendices

Appendix A

The Stannel ISA

A.1 Instruction Encoding

Table A.1 presents the complete instruction set of Stannel. As described in the Chapter 3, a cycle here refers to the length of time to complete a single memory operation.

All instructions, except where noted, encode to a single byte. The upper 4 bits are the *opcode* and the lower 4 bits are the *operand*. In some cases only some of the 4 bits are used for the opcode and in some cases the values used are non-contiguous. This is so that other operations can be added to the ISA at a later date; the assembler includes placeholders for other arithmetic operations such as bit shifting or multiplication.

For communication and scheduling operations we state a minimum bound on the number of cycles required. The minimum bound accounts for the number of cycles required to save the state of the current process to memory, which starts after a single cycle to handle the instruction. After the cycle to process the instruction the core issues a notification of the instruction to the processor itself, which handles communication and scheduling. The number of cycles before this process finishes is non-deterministic as the processor may be handling a message from another core, which must complete before this core's message can be handled.

Instruction	Byte encoding	Cycle count	Effect
`add`	`00`	1	Unsigned addition of top two stack elements
`sub`	`01`	1	Unsigned subtraction
`not`	`08`	1	Bitwise not of top of stack
`or`	`09`	1	Bitwise or of top of stack
`and`	`0A`	1	Bitwise and
`xor`	`0B`	1	Bitwise xor
`test`	`0E`	1	Equivalent to `and` but doesn't push result
`cmp`	`0F`	1	Equivalent to `sub` without pushing result
`push`$_n$	`1`n	1	Pushes the value `000`n
`add`$_n$	`2`n	1	Adds the value `000`n

`push`$_{0abc}$[1]	`3`abc	1	Pushes the value `0`abc
`push`$_{abc0}$[1]	`4`abc	1	Pushes the value abc`0`
`j`**`condition`**	`5`**`condition`**	1/2	Jumps if the condition holds. See Table A.2
`start`	`60`	>6	Start a new process
`end`	`61`	>6	End the current process
`chan`	`62`	>6	Create a new channel
`del`	`63`	>6	Delete channel
`!`	`64`	>6	Send message
`?`	`65`	>6	Receive message
`altstart`	`66`	>6	Enter an alternation in this process
`altwait`	`67`	>6	Wait in an alternation
`altend`	`68`	>6	Leave an in alternation
`enable`	`69`	>6	Enable a channel in an alternation
`disable`	`6A`	>6	Disable a channel in an alternation
`yield`	`6B`	>6	Deschedule this process
`call`	`70`	3	Pushes return address; jumps
`ret`	`71`	2	Pops return address and jumps
`drop`	`80`	1	Drops top of stack
`dup`	`81`	1	Duplicates top stack element
`swap`	`82`	1	Swaps top 2 stack elements
`tuck`	`83`	1	Tucks top 3 stack elements
`rot`	`84`	1	Rotates top 3 stack elements
`get`	`C0`	1	Copies from further down the stack
`get`$_n$	`E`n	2	Copies from further down the stack

Table A.1: Byte encodings for all Stannel instructions.

The pseudo-instruction `nop` is encoded as "jump never". There is special casing within the processor to ensure that this operation does not pop from the stack.

[1] Encodes as 2 bytes

A.2 Flags and Conditions

The processor has four flags,[2] which are set after arithmetic and comparison operations:

- **Zero flag:** true if and only if the result of the operation was zero;
- **Carry flag:** true if and only if the result of the subtraction or comparison operation carried;
- **Overflow flag:** true if and only if the result of the addition operation overflowed;
- **Sign flag:** true if and only if the result of an operation set the most significant bit to 1. Under two's complement arithmetic this is true if the result of the operation was negative.

Condition	Nibble encoding	Function
$=$ | `zero`	`0`	z
$\neq$ | `nonzero`	`1`	$\neg z$
< 0	`2`	s
≥ 0	`3`	$\neg s$
$>_u$	`4`	$\neg c \wedge \neg z$
$\leq_u$	`5`	$c \vee z$
$<_u$	`6`	$\neg c$
$\geq_u$	`7`	c
$>_s$	`8`	$\neg(s \oplus o) \wedge \neg z$
$\leq_s$	`9`	$(s \oplus o) \vee z$
$<_s$	`A`	$\neg(s \oplus o)$
$\geq_s$	`B`	$s \oplus o$
`overflow`	`C`	o
`no-overflow`	`D`	$\neg o$
`never`	`E`	$\bot$
`always`	`F`	$\top$

Table A.2: Encoding of different conditions. $\circ_s$ denotes a signed comparison whilst $\circ_u$ denotes an unsigned comparison. All conditions can be negated by flipping their least significant bit.

[2]These are deliberately the same flags as used in the Intel x86 and Intel x64 architectures [Int18].

Appendix B

Staticks Concrete Grammar

The following serves as an informal summary of Statick's concrete grammar. Our compiler implements a parser for this grammar as a recursive descent parser. The concrete grammar doesn't admit empty terms or definition lists.

```
identifier = [A-Za-z][A-Za-z0-9]+
number     = [0-9]+
name = identifier "_" number                    { Name($1, $3) }
     | identifier                               { Name($1) }

program = definitions                           { $1 }

definitions = definitions definition            { $1 ++ [$2] }
            | definition                        { [$1] }

definition = identifier "=" term                { Definition($1, $3) }

term = term expr                                { $1 ++ [$2] }
     | expr                                     { [$1] }

expr = name                                     { Name($1) }
     | "'" name                                 { Ref($1) }
     | "(" term ")"                             { Anonymous($2) }
     | "if" "(" term ")" "then" "(" term ")" "else" "(" term ")"
                                                { If($3, $7, $11) }
     | "while" "(" term ")" "do" "(" term ")"   { While($3, $7) }
     | "repeat" "_" number "(" term ")"         { Repeat_$3($5) }
     | "repeat" "(" term ")"                    { Repeat_∞($3) }
     | "[" arms "]"                             { Alternation($2) }

arms = arms arm                                 { $1 ++ [$2] }
     | arm                                      { [$1] }

arm = number "->" term                          { Arm($1, $3) }
```

Appendix C

Statick Typing Rules

We present the types of all Statick standard library functions, and typing rules for all expressions and terms. All standard library functions and user defined functions are present in a global environment Γ that maps names to types. The following rules do not account for recursive definitions; we describe our handling of them in Section 4.4.

Note that where we use stack, type, and channel use variables these are fresh in each of their occurrences (so the α and S that occur in the type of `dup` are distinct from the α and S that occur in the type of `drop`).

The standard library functions have the following types:

$$
\begin{aligned}
\Gamma \vdash \texttt{true} &:: S \to S \times \mathbb{B} \\
\Gamma \vdash \texttt{false} &:: S \to S \times \mathbb{B} \\
\Gamma \vdash \texttt{swap} &:: S \times \alpha \times \beta \to S \times \beta \times \alpha \\
\Gamma \vdash \texttt{dup} &:: S \times \alpha : \text{Duplicable} \to S \times \alpha \times \alpha \\
\Gamma \vdash \texttt{tuck} &:: S \times \alpha \times \beta \times \gamma \to S \times \beta \times \gamma \times \alpha \\
\Gamma \vdash \texttt{rot} &:: S \times \alpha \times \beta \times \gamma \to S \times \gamma \times \alpha \times \beta \\
\Gamma \vdash \texttt{toInt} &:: S \times \alpha : \text{IntLike} \to S \times \alpha \times \mathbb{N} \\
\Gamma \vdash + &:: S \times \mathbb{N} \times \mathbb{N} \to S \times \mathbb{N} \\
\Gamma \vdash - &:: S \times \mathbb{N} \times \mathbb{N} \to S \times \mathbb{N} \\
\Gamma \vdash \{>, <, =, \neq, \geq, \leq\} &:: S \times \mathbb{N} \times \mathbb{N} \to S \times \mathbb{B} \\
\Gamma \vdash \texttt{and} &:: S \times \mathbb{B} \times \mathbb{B} \to S \times \mathbb{B} \\
\Gamma \vdash \texttt{or} &:: S \times \mathbb{B} \times \mathbb{B} \to S \times \mathbb{B} \\
\Gamma \vdash \texttt{not} &:: S \times \mathbb{B} \to S \times \mathbb{B} \\
\Gamma \vdash \texttt{apply} &:: S \times (S \to S') \to S' \\
\Gamma \vdash \texttt{del} &:: S \times \texttt{chan}(0, \text{Rx}, \alpha) \to S
\end{aligned}
$$

Certain functions in the standard library can be parameterised by a number. The types for these functions are created lazily in the compiler.

$$\begin{aligned}
\Gamma \vdash n &:: S \to S \times \mathbb{N} \\
\Gamma \vdash ? &:: S \times \mathsf{chan}(n+1, \mathrm{Rx}, \alpha) \to S \times \mathsf{chan}(n, \mathrm{Rx}, \alpha) \times \alpha \\
\Gamma \vdash ! &:: S \times \mathsf{chan}(n+1, \mathrm{Tx}, \alpha) \times \alpha \to S \times \mathsf{chan}(n, \mathrm{Tx}, \alpha) \\
\Gamma \vdash \mathtt{@}_n &:: S \times \alpha_n : \mathrm{Duplicable} \times \alpha_{n-1} \times \cdots \times \alpha_0 \to S \times \alpha_n \times \alpha_{n-1} \times \cdots \times \alpha_0 \times \alpha_n \\
\Gamma \vdash ?_n &:: S \times \mathsf{chan}(k+1, \mathrm{Rx}, \alpha_n) \times \alpha_{n-1} \times \cdots \times \alpha_0 \to S \times \mathsf{chan}(k, \mathrm{Rx}, \alpha_n) \times \alpha_{n-1} \times \cdots \times \alpha_0 \times \alpha_n \\
\Gamma \vdash !_n &:: S \times \mathsf{chan}(k+1, \mathrm{Tx}, \alpha_n) \times \alpha_{n-1} \times \cdots \times \alpha_0 \times \alpha_n \to S \times \mathsf{chan}(k, \mathrm{Tx}, \alpha_n) \times \alpha_{n-1} \times \cdots \times \alpha_0 \\
\Gamma \vdash \mathsf{chan} &:: S \to S \times \mathsf{chan}(\infty, \mathrm{Rx}, \alpha) \times \mathsf{chan}(\infty, \mathrm{Tx}, \alpha) \\
\Gamma \vdash \mathsf{chan}_n &:: S \to S \times \mathsf{chan}(n, \mathrm{Rx}, \alpha) \times \mathsf{chan}(n, \mathrm{Tx}, \alpha) \\
\Gamma \vdash \mathsf{proc}_n &:: S \times \alpha_n \times \alpha_{n-1} \times \cdots \times \alpha_1 \times (S' : \mathrm{MustBeBase} \times \alpha_n \times \alpha_{n-1} \times \alpha_1 \to S'' : \mathrm{NCoU}) \to S
\end{aligned}$$

In the above NCoU denotes 'no consumable or undroppable types'.

We assume the presence of unification functions $\gamma : T \times T \to \mathcal{U}$ and $\gamma : S \times S \to \mathcal{U}$ as described in Section 4.4. Which unification function is used can be derived from context. If the unification function fails to find a most general unifier then the type check algorithm terminates.

$$\frac{}{\Gamma \vdash t : \Gamma[t]}\ \textsc{Name}$$

$$\frac{\Gamma \vdash t_1 :: S_1 \to S_2 \qquad \Gamma \vdash t_2 :: S_3 \to S_4}{\Gamma \vdash t_1\, t_2 :: \gamma(S_2, S_3)(S_1 \to S_4)}\ \textsc{Application}$$

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash \mathsf{Ref}(x) :: S \to S \times T}\ \textsc{Reference}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \mathsf{Anonymous}(t) :: S \to S \times T}\ \textsc{Abstraction}$$

$$\frac{\Gamma \vdash c :: S \to S' \times \mathbb{B} \qquad \Gamma \vdash t :: S' \to S'' \qquad \Gamma \vdash f :: S' \to S''}{\Gamma \vdash \mathsf{If}(c, t, f) :: S \to S''}\ \textsc{If-Expression}$$

$$\frac{\Gamma \vdash c :: S \to S \times \mathbb{B} \qquad \Gamma \vdash b :: S \to S}{\Gamma \vdash \mathsf{While}(c, b) :: S \to S}\ \textsc{While-Expression}$$

$$\frac{\Gamma \vdash t :: S \to S}{\Gamma \vdash \mathsf{Repeat}_\infty(t) :: S \to \bot_S}\ \textsc{Infinite-Repeat-Expression}$$

$$\frac{\Gamma \vdash t :: S \to S' \qquad \Gamma \vdash \underbrace{t\, t \ldots t}_{n \text{ applications}} :: S \to S''}{\Gamma \vdash \mathsf{Repeat}_n(t) :: S \to S''}\ \textsc{Finite-Repeat-Expression}$$

In the following rule, let $U = \gamma(S, S'' \times \mathsf{chan}(k, \mathrm{Rx}, \alpha) \times \alpha_n \times \alpha_{n-1} \times \cdots \times \alpha_1)$. Note that this unifier is applied independently to ensure that the input type of the channel requires that we have one use available.

$$n \in \mathbb{N}\ \frac{\Gamma \vdash t :: S \times \alpha \to S'}{\Gamma \vdash n \to t :: U(S'') \times \mathsf{chan}(k+1, \mathrm{Rx}, U(\alpha)) \times U(\alpha_n \times \alpha_{n-1} \times \cdots \times \alpha_1) \times U(\alpha \to S')}\ \textsc{Arm}$$

In the final rule we find a unifier for the output of every single arm; we can apply the unifier to any of the premise types.

$$n > 0\ \frac{\Gamma \vdash t_1 :: A_1 \to B_1 \qquad \Gamma \vdash t_2 :: A_2 \to B_2 \qquad \ldots \qquad \Gamma \vdash t_n :: A_n \to B_n}{\Gamma \vdash \mathsf{Alternation}(t_1, t_2, \ldots, t_n) :: \gamma(B_1, B_2, \ldots, B_n)(A_1 \to B_1)}\ \textsc{Alternation}$$

Appendix D

Statick Examples

D.1 A recursive Fibonacci function

```
fib =
  if (@0 0 ==) then ()
  else (
    if (@0 1 ==) then ()
                 else (
                   @0 1 - fib swap 2 - fib +
                 )
  )
```

Figure D.1: Statick code for a simple Fibonacci function. The type of the function is $S \times \text{int} \to S \times \text{int}$.

```
f_fib:
  0 get         -- Encodes to a single byte, equivalent to dup
  0 cmp
  l_2 jneq
  ret
l_2:
  0 get
  1 cmp
  l_5 jneq
  ret
l_5:
  0 get 1 - f_fib call
  swap  2 - f_fib call
  +
  ret
```

Figure D.2: The compiled Stannel Assembly for Figure D.1.

D.2 Sending and receiving values forever

```
main = chan 'sender proc_1 repeat (? drop)
sender = repeat (1 !)
```

Figure D.3: The first process repeatedly listens on the same channel, and after receiving a value immediately drops it. The second process always sends the value 1. The type of `main` is $S \to \bot_S$.

```
f_main:
  chan dup   -- The hardware dup instruction pushes one copy of the pointer
  f_sender 1 start
l_1:
  ? drop
  l_1 j
f_sender:
  1 !
  f_sender j
```

Figure D.4: The compiled Stannel Assembly for Figure D.3.

D.3 Listening for values in an alternation

```
main = chan 'p1 proc_1 chan 'p2 proc_1 repeat ([ @0 -> drop | @1 -> drop])
p1 = repeat (0 !)
p2 = repeat (true !)
```

Figure D.5: A Statick program that creates 2 channels and 2 processes, and then repeatedly listens on the two in an alternation. Each alternation arm must leave the stack in the same state, and as the alternation is used in an infinitely repeat looping, each channel must receive an infinite number of values. The type of the first channel is inferred as $\mathsf{Chan}(\infty, \mathsf{Tx}, \mathsf{int})$ because it is passed to a function that infinitely sends integers, whereas the second is $\mathsf{Chan}(\infty, \mathsf{Tx}, \mathsf{bool})$. Alternation arms do not have to use channels of the same type, so long as they all leave the stack in a state with the same types.

```
f_main:
  chan dup f_p1 1 start
  chan dup f_p2 1 start
l_1:
  altstart
    0 get enable
    1 get enable
  altwait
    l_2 1 get disable
    l_3 2 get disable
  altend
l_2:
  drop
  l_1 j
l_3:
  drop
  l_1 j

f_p1:
  0 !
  f_p1 j

f_p2:
  1 !
  f_p2 j
```

Figure D.6: The compiled Stannel Assembly for Figure D.5.

Bibliography

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

[Bak92] H. G. Baker. "Lively Linear Lisp: "Look Ma, No Garbage!"" In: *ACM SIGPLAN Notices* 27 (Aug. 1992), pp. 89–98. DOI: 10.1145/142137.142162.

[Bak94] H. G. Baker. "Linear Logic and Permutation Stacks —- the Forth Shall Be First". In: *SIGARCH Comput. Archit. News* 22.1 (Mar. 1994), pp. 34–43. ISSN: 0163-5964. DOI: 10.1145/181993.181999. URL: http://doi.acm.org/10.1145/181993.181999.

[Ber+17] J. Bernardy et al. "Linear Haskell: practical linearity in a higher-order polymorphic language". In: *CoRR* abs/1710.09756 (2017). arXiv: 1710.09756. URL: http://arxiv.org/abs/1710.09756.

[BO16] R. E. Bryant and D. R. O'Hollaron. *Computer Systems: A Programmer's Perspective*. 3rd ed. Pearson, 2016.

[Bra18] E.C. Brady. *Type-driven Development of Concurrent Communicating Systems*. 2018. URL: https://journals.agh.edu.pl/csci/article/view/1413.

[Bra19] E. Brady. *Idris: A Langauge with Dependent Types*. 2019. URL: https://www.idris-lang.org.

[Bro81] L. Brodie. *Starting Forth*. 1st ed. 1981.

[Car87] L. Cardelli. "Basic Polymorphic Typechecking". In: *Science of Computer Programming* (Apr. 1987).

[CH00] K. Claessen and J. Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP* 46 (Jan. 2000). DOI: 10.1145/1988042.1988046.

[Chi18] D. Chisnall. *C Is Not a Low-level Language*. 2018. URL: https://queue.acm.org/detail.cfm?id=3212479.

[Den18] T. Denney. "A Just-In-Time compiler for the BBC micro:bit". University of Oxford, 2018.

[Dig08a] C. Diggins. *Simple Type Inference for Higher-Order Stack-Oriented Languages*. 2008.

[Dig08b] C. Diggins. *Typing Functional Stack-Based Languages*. May 2008.

[Dig18] C. Diggins. *Cat Programming Language*. 2018. URL: https://github.com/cdiggins/cat-language.

[DM82] L. Damas and R. Milner. "Principal type-schemes for functional programs". In: *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (Jan. 1982), pp. 207–212. DOI: 10.1145/582153.582176.

[Fog18] A. Fog. *Optimizing subroutines in assembly language: An optimization guide for x86 platforms*. 2018. URL: https://www.agner.org/optimize/optimizing_assembly.pdf.

[Gal14] A. Gallant. *QuickCheck for Rust (with shrinking)*. 2014. URL: https://github.com/BurntSushi/quickcheck.

[Gis18] D. Gisselquist. *Aggregating verified modules together*. 2018. URL: http://zipcpu.com/formal/2018/04/23/invariant.html.

[GR+16] T. Gibson-Robinson et al. *Concurrency Lecture Notes*. 2016.

[Han12] S. Hangaslammi. *Concatenative, Row-Polymorphic Programming in Haskell*. 2012. URL: https://github.com/leonidas/codeblog/blob/master/2012/2012-02-17-concatenative-haskell.md.

[Hen98] M. Hendrix. *iForth*. 1998. URL: http://home.iae.nl/users/mhx/i4threads.html.

[HH06] A. J. Harris and J. R. Hayes. "Functional Programming on a Stack-Based Embedded Processor". In: *2nd IEEE International Conference on Space Mission Challenges for Information Technology* (2006).

[Hin69] R. Hindley. "The Principal Type-Scheme of an Object in Combinatory Logic". In: *Transactions of the American Mathematical Society* 146 (1969), pp. 29–60. ISSN: 00029947. URL: http://www.jstor.org/stable/1995158.

[Hoa03] C. A. R. Hoare. *Communicating Sequential Processes*. 2003.

[Hyd95] D. C. Hyde. *Introduction to the Programming Language Occam*. 1995.

[Jal17] G.A. Jaloyan. "Safe Pointers in SPARK 2014". In: *CoRR* abs/1710.07047 (2017). arXiv: 1710.07047. URL: http://arxiv.org/abs/1710.07047.

[Jou+17] N. P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *CoRR* abs/1704.04760 (2017). arXiv: 1704.04760. URL: http://arxiv.org/abs/1704.04760.

[Ker09] A. Ker. *Lambda Calculus and Types Lecture Notes*. 2009.

[Kle17a] R. Kleffner. *MiniJoy*. Mar. 2017. URL: https://github.com/nuprl/hopl-s2017/blob/master/type-inference-for-stack-languages/minijoy.pdf.

[Kle17b] R. Kleffner. *Type Inference for Stack-based Languages*. Mar. 2017. URL: http://prl.ccs.neu.edu/blog/2017/03/10/type-inference-in-stack-based-programming-languages/.

[Koz11] D. Kozen. *Type Inference and Unification*. 2011. URL: https://www.cs.cornell.edu/courses/cs3110/2011sp/Lectures/lec26-type-inference/type-inference.htm.

[Kut02] T. Kutsia. "Pattern Unification with Sequence Variables and Flexible Arity Symbols". In: *Electronic Notes in Theoretical Computer Science* (2002).

[Lev09] D. Levinthal. *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors*. 2009. URL: https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

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

[May87] D. May. *Occam 2 language definition*. 1987.

[Mil78] R. Milner. "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348 –375. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(78)90014-4. URL: http://www.sciencedirect.com/science/article/pii/0022000078900144.

[Mor+98a] G. Morrisett et al. "From System F to Typed Assembly Language". In: *Twenty-Fifth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Jan. 1998).

[Mor+98b] G. Morrisett et al. "Stack-Based Typed Assembly Language". In: *Lecture Notes in Computer Science*. Ed. by X. Leroy and A. Ohori. 1998.

[PJ+98] S. Peyton Jones et al. *Haskell 98*. 1998. URL: https://www.haskell.org/onlinereport/.

[Pou87] D. Pountain. *A tutotorial introduction to OCCAM programming*. 1987.

[Pur12] J. Purdy. *Why Concatenative Programming Matters*. 2012. URL: http://evincarofautumn.blogspot.com/2012/02/why-concatenative-programming-matters.html.

[Pur17] J. Purdy. *Kitten Programming Language*. 2017. URL: https://github.com/evincarofautumn/kitten.

[Qia+07] X. Qian et al. "Optimized Register Renaming Scheme for Stack-Based x86 Operations". In: *Architecture of Computing Systems - ARCS 2007*. Ed. by Paul Lukowicz, Lothar Thiele, and Gerhard Tröster. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 43–56. ISBN: 978-3-540-71270-1.

[Rog16] A. Rogers. *Computer Architecture Lecture Notes*. 2016.

[Rog17] A. Rogers. "Stack: a stack-based virtual machine for physical devices". 2017.

[SBY19] A. Scalas, E. Benussi, and N. Yoshida. *Effpi: Concurrent Programming with Dependent Behavioural Types*. 2019. URL: https://popl19.sigplan.org/event/beat-2019-papers-effpi-concurrent-programming-with-dependent-behavioural-types.

[Sel07] D. Selwood. *The Inmos Legacy*. 2007. URL: http://www.inmos.com/inmos_legacy.html.

[Spi18] M. Spivey. *Compilers Coursebook*. 2018.

[Thu08] M. von Thun. *Mathematical foundations of Joy*. 2008. URL: http://www.kevinalbrecht.com/code/joy-mirror/j02maf.html.

[TP11] J. A. Tov and R. Pucella. "Practical Affine Types". In: *ACM Symposium on Principles of Programming Languages (POPL)*. 2011.

[Ver17] T. Verbeure. *BlackIce-II*. 2017. URL: https://github.com/mystorm-org/BlackIce-II.

[Ver18] T. Verbeure. *Fast SRAM Example*. 2018. URL: https://github.com/mystorm-org/BlackIce-II/tree/master/examples/sram.

[Ver19] Veripool. *Verilator*. 2019. URL: https://www.veripool.org/wiki/verilator.

[Wol18a] C. Wolf. *Project IceStorm*. 2018. URL: http://www.clifford.at/icestorm/.

[Wol18b] C. Wolf. *Yosys Open Synthesis Suite*. 2018. URL: https://github.com/YosysHQ/yosys.

[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.

[ARM18] ARM Limited. *Cortex-M35P: A tamper-resistant Cortex-M processor with optional software isolation using TrustZone for Armv8-M*. 2018. URL: https://developer.arm.com/ip-products/processors/cortex-m/cortex-m35p.

[Con17] Concatenative Languages Wiki Authors. *Concatenative Languages*. 2017. URL: http://concatenative.org/wiki/view/Concatenative\%20language.

[Goo19] Google. *A Tour of Go: Channels*. 2019. URL: https://tour.golang.org/concurrency/2.

[GTK19] GTKWave Authors. *GTKWave*. 2019. URL: http://gtkwave.sourceforge.net.

[Ica18] Icarus Verilog Authors. *Icarus Verilog*. 2018. URL: http://iverilog.icarus.com.

[Inm85] Inmos Limited. *The Implementation of OCCAM on the IMS T414*. 1985.

[Inm87] Inmos Limited. *Transputer Architecutre*. 1987. URL: http://www.transputer.net/fbooks/tarch/tarch.html.

[Inm88] Inmos Limited. *Transputer Instruction Set: A Compiler Writer's Guide*. Prentice Hall, 1988.

[Int18] Intel. *Intel 64 and IA-32 Architectures Developer's Manual*. Vol. 1. 2018.

[Lat16] Lattice Semiconductor. *Memory Usage Guide for iCE40 Devices*. 2016. URL: https://www.latticesemi.com/-/media/LatticeSemi/Documents/ApplicationNotes/MO/MemoryUsageGuideforiCE40Devices.ashx?document_id=47775.

[Lat17] Lattice Semiconductor. *iCE 40 LP/HX Data Sheet*. 2017. URL: https://www.latticesemi.com/~/media/LatticeSemi/Documents/DataSheets/iCE/iCE40LPHXFamilyDataSheet.pdf.

[Lat19] Lattice Semiconductor. *iCE 40 LP/HX/LM*. 2019. URL: http://www.latticesemi.com/Products/FPGAandCPLD/iCE40.aspx.

[Mic17] Microsoft. *Common Language Runtime*. Oct. 2017. URL: https://docs.microsoft.com/en-us/dotnet/standard/clr.

[Mic18a] Micro:bit Educational Foundation. *micro:bit*. 2018. URL: https://www.microbit.org.

[Mic18b] Microsoft. *Language Server Protcol Specification*. 2018. URL: https://microsoft.github.io/language-server-protocol/specification.

[Moz18] Mozilla Foundation. *The Rust Programming Language: Using Message Passing to Transfer Data Between Threads*. 2018. URL: https://doc.rust-lang.org/stable/book/ch16-02-message-passing.html.

[Nor14] Nordic Semiconductor. *Nordic nRF51822*. 2014. URL: https://infocenter.nordicsemi.com/pdf/nRF51822_PS_v3.1.pdf.

[Uni05] University of Glasgow. *Data.Ord*. 2005. URL: https://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Ord.html.