Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Sindre Amdal Stephansen

The Challenges of porting Inferno to RISC-V

Master's thesis in Computer Science
Supervisor: Michael Engel
August 2021

NTNU
Norwegian University of Science and Technology

Sindre Amdal Stephansen

The Challenges of porting Inferno to RISC-V

Master's thesis in Computer Science
Supervisor: Michael Engel
August 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Contents

Abstract

The RISC-V processor architecture is rapidly rising in popularity, and there will probably be an explosion of smaller RISC-V computers in the coming years, as sensors, in appliances, and more. Because these kinds of computers do not always have the resources to run an operating system like Linux, the Inferno operating system is an alternative, which, with its networked and distributed nature, could be a perfect match for these kinds of systems.

In this thesis I begin to port Inferno to RISC-V, and identify the challenges of both porting and using the operating system.

The first major challenge was to get the system to a stage where it could boot and handle simple input and output. The second challenge was to make the system more usable by implementing drivers. The last challenge was to implement a Just-in-time compiler, to make the system more responsive.

While not fully usable yet, I have made significant progress in porting Inferno. The operating system boots and launches an interactive shell, in which the user can execute commands. It can output to both a serial port and a screen. I have implemented a Just-In-Time compiler, but there are some bugs which cause complicated programs to crash.

This forms the foundation from which a port of Inferno to real hardware can be built.

Sammendrag

Prosessorarkitekturen RISC-V blir stadig mer populær, og det vil sannsynligvis bli en eksplosjon av små RISC-V maskiner de neste årene, som sensorer, i hvitevarer, og mer. Siden disse typene datamaskiner ikke alltid har ressursene til å kjøre operativsystemer som f.eks. Linux er operativsystemet Inferno et alternativ. Infernos distribuerte og nettverksorienterte design kan passe utmerket for disse typene systemer.

I denne oppgaven begynner jeg på å tilpasse Inferno til å kjøre på RISC-V, og identifiserer utfordringene med å tilpasse og bruke operativsystemet.

Den første store utfordringen var å få systemet til et punkt der det kunne starte opp og håndtere enkel kommunikasjon, i form av tekst. Den andre utfordringen var å gjøre systemet brukbart ved å implementere enhetsdrivere. Den siste utfordringen var å implementere en Just-in-Time kompilator, for å gjøre systemet mer responsivt.

Selv om operativsystemet ikke er helt brukbart ennå har jeg gjort store framskritt. Operativsystemet starter opp og viser en kommandolinje der brukeren kan utføre kommandoer. Tekst kan printes både til en seriell port og til en skjerm. Jeg har implementert en Just-in-Time kompilator, men det er noen problemer som får kompliserte programmer til å krasje.

Dette prosjektet danner grunnlaget for å bruke Inferno på RISC-V maskiner.

1 Introduction

The RISC-V platform is gaining ground in research as well as industrial projects. However, system software support so far is mostly focusing on the well-known large open source operating systems (Linux, BSD) or very tiny embedded real-time kernels. The large systems have now grown too big for many applications on restricted hardware platforms, e.g. running on a small FPGA-based system, whereas the traditional real-time operating systems suffer from a lack of useful network integration, memory protection, or orthogonal concepts of files and file systems, which makes their use in networked settings (IoT, Cloud) more challenging.

Thus, the idea of this project is to cover the middle ground by porting the open source Inferno operating system from Bell Labs to RISC-V. This system is already highly portable, but a RISC-V port is missing. Inferno is especially interesting since it is well documented and the low complexity of the system (compared to e.g. Linux) makes it very suitable to work on in the context of a student project.

During this project I started the work to create a port of Inferno to RISC-V, running under QEMU. I managed to get it to compile, print and receive input through UART. It enables and handles traps, schedules processes, starts the virtual machine, and launches an interactive shell which the user can execute commands from. I started to implement a Just-in-Time compiler, but there are bugs which causes crashes with complex programs.

The source code of the project is available at `https://github.com/kalkins/inferno-os/tree/riscv`. The code for the Just-in-Time compiler is at a separate branch, at `https://github.com/kalkins/inferno-os/tree/riscv-jit`.

This report is structured as follows: Section 2 gives an overview over the technologies used in this project. Section 3 goes through the development step by step. Section 4 covers major problems I encountered, and how I dealt with them. Section 5 summarizes the current state of the port. Section 6 discusses what remains in order to have a working port, and how this port could be used in practice.

2 Background

2.1 Plan9

Plan9 from Bell Labs, commonly shortened to Plan9, is a distributed operating system designed to solve some of the problems with UNIX-based workstations [18]. The operating system is designed to be distributed over a network of smaller workstations, giving each the full power of the network. Instead of maintaining UNIX compatibility, Plan9 kept the ideas that worked and redesigned the rest. Plan9 has a new suite of compilers, new libraries, and polished suite of tools.

Plan9 is built around the UNIX concept that everything is a file, and extends it. Most system resources are represented as files in the filesystem, and the files from other computers on the network are seamlessly available in the same filesystem. Because of this each machine can be responsible for a class of services which are made available through files, and any computer on the network can use those services as if they were hosted locally.

Plan9 also incorporated a concept of per-process name spaces, which means that each process has their own view of the filesystem. This is used, for example, by the graphical interface: When a process wants to display something on screen it writes to the `/dev/bitblt` file. The window system process can replace this file in the name space of its subprocesses and therefore intercept all writes. When the subprocess writes to `/dev/bitblt`, believing that it writes to the whole screen, the window system receives the request, translates the coordinates to within the window given to the subprocess, and writes to the `/dev/bitblt` file in its own name space. This provides a simple way to encapsulate processes and build nested structures.

2.2 Inferno

Inferno is a distributed operating system based on Plan9 which focuses on portability and versatility, intended to be used for phones, TVs, and personal computers [8]. Inferno applications are written in the Limbo language and are compiled to byte-code which runs on a virtual machine, called Dis. The virtual machine was designed to be close to modern processor architectures at the time, and to make Just-in-time compilation fast and easy. The designers of the virtual machine claim that the JIT compiled code is 30-50% slower than native C [26].

2.3 RISC-V

At the time of writing the dominating ISAs, x86 for personal computers and servers, and ARM for embedded systems and phones, are proprietary. For x86 this has resulted in there being only two CPU manufacturers for mid- to high-end systems. For ARM, manufacturers must pay a licensing fee to use the design, which increases manufacturing cost, and there is little room for adapting the design to the rest of the hardware [1]. These designs are also often held back by the requirement of backwards-compatibility.

RISC-V, on the other hand, is a modern, free, open-source instruction-set architecture, which means that anyone can design a processor that fits the specification without paying licensing fees, and the design can be adapted to the hardware. Small embedded devices may implement the ISA in a cheap, straight-forward way, while desktop devices can use advanced techniques to get as much performance as possible.

The full ISA specification can be found in Waterman and Asanović [22, 23].

2.3.1 ISA extensions

One of the most interesting things about RISC-V is that the instruction-set is modular. The specification defines a few base ISAs, and several extensions that may or may not be dependent on other extensions. This allows hardware manufacturers to implement only the functionality that is needed, keeping the hardware simple for embedded devices, while allowing power and functionality to higher-end devices.

The width of registers is defined by XLEN, which is set by the base ISA. Most computational instructions are defined by this value, and therefore automatically use the available width on the platform. Platforms may allow XLEN to be changed at runtime.

The most basic ISA is RV32I, which uses `XLEN=32`, 32 registers, and defines common instructions like addition, move, load, store and branches. RV64I is another base ISA which builds on RV32I by keeping the instructions but changing XLEN to 64 and defining new instructions for 32-bit values. RV128I similarly changes XLEN to 128 and adds instructions for 64-bit values. RV32E, a base ISA with 16 registers designed for embedded systems, is currently in a draft stage.

The fact that the value of XLEN changes how instructions behave means that a program compiled for RV32I can be loaded on a 64-bit or 128-bit system and use the whole register width automatically. This can cause problems if the developer designs the code around 32-bit registers, especially if the code is designed to overflow. However, if the developer designs the code to work on both 32-bit and 64-bit platforms the full available width can be utilized without having separate versions for each value of XLEN.

Instruction-set extensions can be added to any base ISA in any configuration, as long as their dependencies are also included. The most notable extension is perhaps the **F** extension which adds 32 registers for single-precision floats, and instructions to handle them. The **D** and **Q** extensions build upon this the same way as RV64I and RV128I, adding support for double and quad precision floats. There is also the **M** extension for multiplication and division, **A** for atomic instructions, and **C** for compressed instructions, which provides shorter variants of common instructions.

2.3.2 Privilege levels

The RISC-V specification defines multiple privilege levels, commonly called modes, in which code can be executed. The current mode determines which privileged instructions are available and how traps are handled. A higher mode can fully control a lower mode, providing security and functionality to operating systems and hypervisors.

Code running in a mode can make it impossible for code in lower modes to know which mode they are running in. When the lower code tries to do an operation that requires a higher privilege level, the upper code can emulate the operation.

There are three modes currently defined:

- Machine mode is the only mandatory mode defined by the specification, and it is the highest possible mode with full access to the platform. However, if only machine mode is available none of the benefits of privilege modes are available.
- User mode is an optional mode, and is always the lowest mode. It is used to run insecure user code, with higher modes granting protection.
- Supervisor mode is an optional mode that can be added between machine mode and user mode. It can be used for running operating systems with the bootloader in machine mode, the OS in supervisor mode, and user applications in user mode.

2.3.3 CSR - Control and Status Register

The RISC-V Zicsr extension defines a separate address space that can contain 4096 Control and Status registers (CSRs), and the instructions to use them [22, Chapter 9]. At the time of writing, over 200 CSRs have been defined.

The CSRs are divided into machine mode, supervisor mode and user mode and can be used to read platform information or enable and handle traps, timers, memory protection and virtual address translation for the different modes. CSRs also include information about XLEN and available extensions, so software can adapt to the platform at runtime.

When referring to a CSR independent of mode I use the format `xstatus`, which refers to `mstatus`, `sstatus`, and `ustatus`.

For the full list and description of CSRs, see Waterman and Asanović [23, Chapter 2, 3.1, 4.1].

2.3.4 SBI - RISC-V Supervisor Binary Interface

The Supervisor Binary Interface (SBI) is a standardized interface between software running in supervisor mode, usually operating systems or unikernels, and software running in higher modes, usually bootloaders or hypervisors. The SBI interface abstracts platform specific functionality, so that programs can be ported to all RISC-V implementations.

The SBI specification currently defines several extensions which the bootloader can offer, like setting timers, sending messages between harts, controlling a hart's state, performance monitoring and resetting the system. Earlier versions of the specification defined functions for reading and writing to a console, but these are now deprecated.

SBI functions are called using a standardized calling convention, which is a hybrid of the RISC-V and Linux calling conventions. Like the standard RISC-V convention, the registers `a0` to `a7` are used for arguments, but like in Linux the `a7` register is used for the ID of the extension that is called. Register `a6` can be used for the ID of the function, if the extensions has multiple functions. The call itself is made with an `ECALL` instruction, which causes an exception in the higher modes. The return value is placed into `a1`, with `a0` indicating whether an error occurred.

The specification is still a draft in version 0.3, but it has been implemented by some bootloaders (see section 3.5). It can be found in Dabbelt and Patra [5].

2.3.5 Traps

Traps cause the currently executed code to be stopped, and control is transferred to a trap handler, usually in a higher mode. In RISC-V interrupts are traps that are used as notifications from instructions or devices. Exceptions are errors and environment calls.

Traps are an essential part of operating systems, for system calls, process scheduling, and error handling. In RISC-V traps are layered by mode: First, traps from any mode are sent to machine mode. The program running in machine mode can, before the trap, choose to delegate some or all traps occurring in supervisor or user mode to the program running in supervisor mode. Likewise, user mode traps can be delegated to the program running in user mode from supervisor mode.

Each mode has separate CSRs for enabling and handling traps. There are three classes of interrupts: software interrupts, timer interrupts, and external interrupts. After enabling these, interrupts also have to be enabled globally for the current mode `y` in the `ystatus` CSR. Exceptions can not be disabled, only delegated to a lower mode. When a trap is triggered and sent or delegated to mode `y` the trap handler at the address stored in `ytvec` is called. The specific cause of the trap is stored in the `ycause` CSR.

3 Implementation

3.1 RISC-V compiler

Before I could write any code for the port, I had to find a compiler which was compatible with the Plan9/Inferno compiler architecture, and could compile to RISC-V. Luckily, Richard Miller had already developed and published a RISC-V compiler for Plan9 by the time I started this project. The compiler source can be found in Miller [16].

I began the project by integrating Richard Miller's compiler into Inferno. Plan9 and Inferno have quite similar compiler structure, so this was easy to do. I came across some bugs in the compiler which I fixed as best I could.

After the compiler was integrated I started writing the architecture-specific code necessary for kernel and virtual machine functions, as described in section 3.2.

However, halfway through the project Richard Miller announced that he was working on a new improved compiler, with 64-bit support and more ISA extensions. This new compiler solved all the issues I was having with the old compiler. Richard Miller even added the architecture-specific code for Inferno, which replaced some of my attempts. See section 3.2 for more details. The compiler source can be found in Miller [17]. The compiler was later merged into the Inferno codebase [15].

This new compiler supports both the RV32I and RV64I base extensions, and the **I**, **M**, **A**, **F**, **D**, and **C** ISA extensions. Of these extensions only **C**, for compressed instructions, can be disabled. For the rest, if the platform does not support them the instructions either have to be avoided, or they can cause traps and be emulated in software.

For this project I used the 32-bit compiler because Inferno is a 32-bit operating system.

3.2 Architecture-specific code structure

Some architecture-specific code is necessary for the kernel and Dis virtual machine to run on the hardware. Some of these files were provided by Richard Miller as part of his new compiler, and some I have implemented myself. Most implementations are similar to that of other architectures.

- `Inferno/riscv/include`
  These are architecture-specific header files which are used across the kernel.
  - `lib9.h`
    This file includes other header files.
  - `u.h`
    This file defines type aliases and floating point configuration constants.
  - `ureg.h`
    This file defines the `Ureg` struct, which is used to store register values.
- `libinterp`
  This folder contains code for the Dis virtual machine. See section 3.16.
  - `comp-riscv.c`
    This file contains the implementation of the JIT compiler for RISC-V.
  - `das-riscv.c`
    This file contains the implementation of a RISC-V disassembler, which is used to debug the JIT compiler.
- `libkern`
  This folder contains code for kernel libraries. These files were provided by Richard Miller.
  - `mkfile-riscv`
    Specifies source files for the architecture.
  - `frexp-riscv.c`
    This file provides functions for double-precision floats.
  - `getfcr-riscv.s`
    This file provides functions for reading and writing to the floating-point control and status register.
  - `memmove-riscv.s`, `memset-riscv.s`, and `strchr-riscv.c`
    These files implement the POSIX functions `memmove`, `memset`, and `strchr` for the architecture.
  - `vlop-riscv.c` and `vlrt-riscv.c`
    These files defines functions for arithmetic operations on integers longer than the platform bit width.
- `utils/libmach`
  These files were provided by Richard Miller. The files with an `i` in the name are for RV32I, and those with a `j` in the name are for RV64I.
  - `uregi.h` and `uregj.h`
    These files define the `Ureg` struct, which is used to store register state.

  - `i.c` and `j.c`
    These files define the RISC-V registers and address space.
  - `idb.c` and `jdb.c`
    These files define a RISC-V specific debugger interface.
  - `iobj.c` and `jobj.c`
    These files provide functions used by the `iar` utility to help it recognize RISC-V object files.

3.3 Choosing a platform

Before any platform-specific code could be written, a platform had to be chosen. There are a few physical RISC-V processors available, but physical hardware can be hard to debug.

Instead, I chose to use QEMU. QEMU is a machine emulator which supports many architectures, and can emulate existing physical platforms [3]. It has support for the RV32I and RV64I base ISAs with all current ISA extensions, and machine, supervisor, and user mode. QEMU also has integrated GDB support, to make debugging easier, and it can emulate many input, storage, networking, and graphical devices.

3.4 Platform-specific code structure

The platform-specific code lives in `os/<platform>`, which is `os/qemuriscv` in this case. This code handles initialization of hardware, and provides functions that the rest of the kernel can use, hiding implementation details behind a common interface.

Because of the standardized nature of RISC-V, most of this code can be used for any RISC-V platform. However, at the moment only the QEMU platform has been added, so the code lives in that folder.

Most of the information about platform porting of Inferno comes from a series of blogposts by LynxLine Labs [10].

3.4.1 Header files

The kernel code often includes specific header files, expecting them to be defined in the platform-specific folder and provided to them through the linker. This means that the code is able to adapt better to the platform, but also that a lot of functions have to be defined before a minimal version of the kernel can be compiled.

Here is a list of the header files that had to be added to compile the kernel, and a description of what they provide:

- `mem.h`
  This file defines the memory map of the platform, usually with macros. For more details see section 3.6.
- `dat.h`
  This file defines platform-specific data structures like locks, labels, and machine configuration.
- `fns.h`
  This file defines most platform-specific functions that other parts of the kernel need.

3.4.2 Functions

The common part of the Inferno kernel declares and uses several functions which are not implemented, which have to be implemented by the platform-specific part of the kernel. Here is an overview over those functions, and what they do:

- `int setlabel(Label*)` and `void gotolabel(Label*)`
  These functions handle labels, which contain a program counter and a stack pointer. `setlabel` returns a label with the current program counter and stack pointer values, while `gotolabel` writes the stack pointer to the stack pointer register and jumps to the address in the labels program counter.

- `ulong getcallerpc(void*)`
  This function returns the address of the instruction that called the function.

- `int _tas(int*)`
  This function does a test-and-set, which is a simple atomic operation: It writes a 1 to the given address, and returns the previous value at that address. This can be used to create locks: When a 0 is returned the lock has been acquired, and a 0 can be written to release the lock. The RISC-V **A** extension includes an atomic swap instruction which makes this implementation very easy.

- `int splhi(void)`, `int spllo(void)`, `void splx(int)`, `void splxpc(int)`, and `int islo(void)`
  These functions enable, disable, or toggle interrupts [14]. `islo` returns non-zero if interrupts are enabled. These can be implemented by setting, clearing, or toggling the supervisor mode interrupts in `sstatus`.

- `void kprocchild(Proc*, void (*)(void*), void*)`
  This function configures a kernel process with a stack.

- `int segflush(void*, ulong)`
  This function flushes a region to memory and invalidates the region in the instruction cache.

- `void idlehands(void)`
  This function is called when process runner has nothing to do. It does not have to do anything.

- `void setpanic(void)`
  This function is called to prepare for a panic, if necessary.

- `void dumpstack(void)`
  This function dumps debug information about the stack to the user. It is only meant to help with debugging, and can be empty.

- `Timer* addclock0link(void (*)(void), int)`
  This function sets a given function to be called after a given delay.

- `void clockcheck(void)`
  This function is called to reset the watchdog timer, if necessary.

- `void FPinit(void)`, `void FPsave(void*)`, and `void FPrestore(void*)`
  These functions are called from the Dis virtual machine to enable or disable floating point operations.

- `void exit(int)`, `void reboot(void)`, and `void halt(void)`
  These functions respectively shut down, reboot, and send the system into an infinite loop.

3.4.3 The configuration file

Each platform must have a configuration file, which describes which parts of the OS should be compiled, global configuration variables, and which files and folders should be included in the filesystem. By convention the file has the same name as the platform, without a file extension, so for this platform the configuration file is `os/virtriscv/virtriscv`. The file uses a specific format, and is parsed by a script before compilation. It is divided into sections, where the section name is at the baseline and the contents are indented, one entry per line.

The configuration file is parsed once to import the mkfile dependencies, once to generate a C file which defines the global variables and links device drivers, and once to generate an assembly file and a header file with the contents of the filesystem.

The following sections are common in the configuration files:

- `dev`
  This sections defines device drivers source files to include from the `/os/port/` directory with the `dev` prefix.
- `ip`
  This section defines C files to include from `/os/ip/`, which contain the network stack.
- `lib`
  This section defines libraries to include. These are whole directories at the root level with the `lib` prefix, which are compiled into libraries and linked into the binary.
- `misc`
  This section defines C files to include from the platform directory.
- `mod`
  This section defines Limbo module definitions to include from `module/`.
- `port`
  This section defines C files to include from `os/port/`.
- `code`
  This section consists of C code which declares configuration variables, like enabling the JIT compiler.
- `init`
  This section has only one entry, which defines the initial program to run in the virtual machine. This program will usually set up the system, then start either the shell or the window manager. The entry is the basename of the Limbo program in the `os/init/` folder. The Limbo program will be added as a mkfile dependency, and compiled when changed.
- `root`
  This section defines the filesystem that is included in the binary. Each line specifies a path. If the path ends in a slash it represents a folder which should be present in the filesystem, but does not exist in the local filesystem. If the path does not end with a slash, the file with that path relative to the project root folder is copied into the filesystem, with that path. For example, if the Inferno project root is `/usr/inferno`, and the line `/dis/cd.dis` is in the `root` section, the file `/usr/inferno/dis/cd.dis` will be copied from the local filesystem to `/dis/cd.dis` in the filesystem in the binary.

  The exception is the file `/osinit.dis`, which is copied from the location specified in the `init` section.

This section is mostly used to include essential programs and utilities in the binary. The rest of the programs, and other files, should be on a filesystem that is mounted after boot.

The full configuration file for this project is included in appendix A.

3.4.4 The mkfile

The mkfile defines the build process for the platform. It specifies the target architecture, the name of the configuration file, the platform specific header and source files (which are usually not included in the configuration file), and how the resulting binary is compiled and linked. The full mkfile is included in appendix B.

3.5 OpenSBI

OpenSBI is a bootloader developed by the RISC-V foundation which supports SBI and is included by default by QEMU when using the `virt` machine type. It initializes the machine, switches to supervisor mode, and jumps to a specified address, `0x80400000`, where a binary can be placed to be executed. By passing that address to the linker with the `-T0x80400000` flag and exporting to ELF with the `-H5` flag, the resulting ELF can be passed to QEMU and will be loaded correctly and started by OpenSBI.

For this project calls to OpenSBI will only be used to request timers, because RISC-V timers can only be set from **M** mode, and to shut down the system. The legacy SBI supported console I/O, but this feature is deprecated, and I only used it for debugging other I/O methods.

3.6 Address space

In QEMU RAM starts at address `0x80000000`, with the size being defined by the `-m` command-line parameter. OpenSBI is loaded in at address `0x80000000-0x8001ffff`, and expect the kernel code to be loaded at address `0x80400000`.

There is little documentation about the address space in Inferno, and other implementations are not fully consistent, but it seems like the kernel uses the space below where the kernel code is loaded in, while the user-space uses the space above the kernel. I gave the kernel 8 KiB of stack space from the kernel start at `0x80400000` and downwards. The space from the end of the kernel binary until the end of memory is used for pages for processes. Because the size of RAM can be varied with QEMU, I assume that 128 MiB is available, and the OS will not use more than that. Though Inferno normally does not need that much RAM to run, memory overflow bugs are common during the initial porting process, so it is advantageous to start of with larger RAM sizes. In the future, it might be possible to determine RAM size at runtime and adapt to that.

Supervisor mode does support virtual memory, but I have not used that functionality yet. Because all user processes in Inferno run in a virtual machine, hardware virtual memory is not necessary. However, it would be a useful security measure.

3.7 Using CSRs

Control and Status registers, as mentioned in Section 2.3.3, are used to handle traps, and therefore are vital to an operating system.

The problem with CSRs are that the instructions to operate on them encode both the operation and CSR address. That means that it is impossible to make a generalized function that can do any operation on any CSR at runtime, because the operation and address must be known at compile-time. Instead, separate functions have to be defined for each operation for each CSR.

Some compilers, like GCC, allow these to be implemented in C using inline assembly, but the Plan9 C compiler does not support this, so the functions have to be written in assembly.

Because writing four basically identical functions for a large set of CSR is a boring task, an automatic solution was needed. I created the script `generate-csr.sh` which reads a file `csrregs.h` which contains definitions of CSRs on the form shown in listing 1. The script reads the CSR names and writes function declarations to `csr.h` and implementations to `csr.s`, as shown in listing 2 and 3. These functions return signed numbers because some CSRs have a flag at the MSB position, and the code can check if the CSR value is less than 0 to check the flag regardless of the data width. For example, `xcause` uses the MSB to indicate whether the trap was caused by an interrupt, so the code can simply check whether `xcause` is less than 0 to see if the trap was caused by an error or an interrupt.

When including all currently defined CSRs the resulting code is around 4000 lines (1000 lines of function declarations and 3000 lines of assembly), which compiles to around 3 kilobytes, or 2% of the whole binary. Of course not all CSRs are needed, so the size can be reduced by commenting out sections of `csrregs.h`.

```
#define CSR_ustatus          0x000
#define CSR_uie              0x004
#define CSR_utvec            0x005
```

Listing 1: A snippet from `csrregs.h`

```
long csr_read_ustatus(void);
long csr_write_ustatus(long);
long csr_set_ustatus(long);
long csr_clear_ustatus(long);
```

Listing 2: A snippet from `csr.h`

```
        TEXT csr_read_ustatus(SB), $-4
            CSRRS CSR(CSR_ustatus), R0, R8
            RET

        TEXT csr_write_ustatus(SB), $-4
            CSRRW CSR(CSR_ustatus), R8, R8
            RET

        TEXT csr_set_ustatus(SB), $-4
            CSRRS CSR(CSR_ustatus), R8, R8
            RET

        TEXT csr_clear_ustatus(SB), $-4
            CSRRC CSR(CSR_ustatus), R8, R8
            RET
```

Listing 3: A snippet from `csr.s`

3.8 Handling traps

OpenSBI delegates most traps to supervisor mode, and starts the kernel in supervisor mode. Enabling interrupts when in supervisor mode requires writing the trap handler address to `stvec`, setting the `sie` bit in `sstatus`, and setting the bits corresponding to the desired traps in the `sie` CSR. The trap handler must save all registers, and restore them before returning, to avoid corrupting the state of the interrupted code. Another trap can occur while a trap is being handled, so the registers and stack have to be treated carefully.

QEMU has a separate layer for hardware interrupts through a platform level interrupts controller (PLIC) [4]. This controller is mapped at `0x0c000000`, and handles UART and disk interrupts. Interrupts for those devices are marked as external in the `xcause` CSR, and the PLIC has to be queried to get the exact cause. For UART interrupts are triggered when the input buffer starts being filled or the output buffer is empty.

3.8.1 Listener interface

I implemented a trap listener interface based on the one in the pc port, which allows various parts of the operating system to add and remove trap listeners at any time. There can be multiple listeners for each trap. PLIC interrupts are separated into a separate bus, selected with the `tbdf` argument (name kept for consistency with the pc port).

The interface consists of the following functions:

- `void intrenable(long irq, void (*f)(Ureg*, void*), void* a, int tbdf, char *name)`
  This function enables a trap listener with the given interrupt request number (irq) and bus (tbdf). If it is the only listener for a maskable interrupt, the interrupt is unmasked.

- `int intrdisable(int irq, void (*f)(Ureg *, void *), void *a, int tbdf, char *name)`
  This function disables a trap listener with the given interrupt request number (irq) and bus (tbdf). If it is the only listener for a maskable interrupt, the interrupt is masked.

3.9 Clock and timers

3.9.1 Timers in RISC-V

The RISC-V specification defines a standard way of reading wall-clock time and setting timers. The platform should implement a machine mode accessible memory-mapped register, `mtime`, which ticks up at a fixed rate, though the rate might be different for each platform. There should also be a memory-mapped `mtimecmp` register. A timer interrupt should happen when the value of `mtime` is greater than the value of `mtimecmp` [23, Chapter 3.1.10].

These registers are not accessible from supervisor or user mode. Instead, the current time can be read from the `time` and `timeh` CSRs. These can be implemented to point to `mtime`, or the request can be intercepted and handled in machine mode.

The RISC-V specification does not define a way for lower privilege levels to set timers, instead leaving it up to the machine mode software to define a method for this. The SBI specification defines a method for this with the `void sbi_set_timer(uint64_t stime_value)` in the Timer extension, which allows software in supervisor mode to request a timer interrupt at a given time [5].

3.9.2 Timers in Inferno

Inferno implements most of the functionality for multiplexed timers on a single native timer. The following functions are left to be implemented in the platform-specific code:

- `uvlong fastticks(uvlong *hz)`
  This function returns the current value of the real-time clock, and writes the period of the clock to `hz`.

- `void timerset(uvlong next)`
  This function sets a timer interrupt to trigger when the real-time clock reaches the value of `next`.

- `void clockcheck(void)`
  It is unclear what this function does. It is only called when busy-waiting for locks, and in some platform-specific drivers. All platforms implement it as an empty function. Some implementation comments mention that the function is used to reset watchdog timers.

- `void delay(int milliseconds)` and `void microdelay(int microsecond)`
  These functions busy-waits for a given number of milli- or microseconds.

3.9.3 Implementing the interface

There does not seem to be a standardized way to find the clock period, but through testing I found that the period in QEMU is 10 000 000 Hz. I later verified this in the QEMU source code [19, include/hw/intc/sifive_clint.h, line 57].

In addition to the required functions I implemented the following functions:

- `void clockinit(void)`
  This function enables timer interrupts and calls `timerset` to set a timer infinitely far in the future. It is called during setup before any timers are set.

- `void clockintr(Ureg *ureg, void*)`
  This is the handler for timer interrupts. It sets a new timer infinitely far in the future, and calls `timerintr` function in Inferno.

During testing, I discovered that setting a timer to -1 through SBI immediately caused a timer interrupt, even though the SBI documentation specifies that this is a method to set a timer infinitely far in the future. Through testing, I discovered that setting timers higher than 2^{61} sometimes immediately triggers the timer interrupt. As a temporary workaround, I used the value 2^{60} as infinity, as that is over 3000 years in the future. See section 4.5 for more details.

3.10 UART

The serial port is an essential way for an operating system to communicate with the outside. In QEMU all output from the operating system through the serial port is printed to the screen, and anything the user types in the terminal QEMU is running in is sent through the serial port to the operating system.

QEMU emulates the 16550a UART to handle serial port communication. The UART has eight byte-wide registers which are mapped to the memory addresses `0x10000000-0x10000007`. The pc port of Inferno includes a driver for the 8250 UART line, which supports the 16550a. The driver needed a little configuration for finding the UART port and setting up interrupts, but it mostly worked right out-of-the-box.

The UART port is set up by calling `i8250console` during system initialization, which sets up the FIFO, and configures the UART to use 9600 baud, 8 data bits, 1 stop bit, and no parity. For use with QEMU the configuration has little impact, as the whole system is emulated.

3.11 VIRTIO

While UART is useful for basic input and output, other devices are necessary for a fully usable system. QEMU can emulate a large variety of such devices, which gave me the choice of which drivers I wanted to implement. While the codebase for Inferno includes drivers for several devices I chose not to use them because they are old, possibly unstable, and might have compatibility problems with QEMU.

Instead, I chose to use VIRTIO [21] devices, because VIRTIO uses the same communication protocol for all types of devices, which eases driver development. VIRTIO also performs better than other drivers on QEMU, because it reduces the layers of abstraction between the host and guest systems. The disadvantage of VIRTIO is that it is not implemented on real hardware, and the drivers are therefore only usable for testing or running virtualized systems.

QEMU supports VIRTIO over PCI or memory-mapped IO (MMIO). While the pc port includes a PCI driver which could be adapted for the RISC-V port, I instead decided to use MMIO because of the simplicity of using such an interface. This requires that the VIRTIO addresses and irq numbers are configured at compile-time.

3.11.1 The VIRTIO communication protocol

For VIRTIO over MMIO all VIRTIO devices have a predefined address region. For QEMU these addresses are `0x10001000`, `0x10002000`, up to `0x10008000`, which gives a maximum of 8 VIRTIO devices. The memory region for each device starts with a set of *device registers*, which are used to negotiate device features, setup interrupts, and give the device pointer to the communication queues. After the registers there is a device-specific *configuration space*, which usually contains information about the device.

Data is sent between the driver and the device using *Virtqueues*. Each type of device has a different number of virtqueues for different purposes. This implementation uses *Split Virtqueues*, which separates the buffers the device should read from, and the buffers it should write to. A split virtqueue consists of three ring buffers: the *Descriptor table* containing pointers to buffers and metadata, the *Available Ring* with indexes of descriptors the device should handle, and the *Used Ring* with indexes of descriptors which the device has handled. Often the driver needs to send data and get a response, for which it allocates one descriptor and buffer pair for the device to read, and another pair for the device to write the response to, and sends them together in a descriptor chain. The driver is responsible for allocating the virtqueue and all buffers. The driver usually deallocates the associated buffer after a response.

The VIRTIO specification often describes messages as a single structure. However, such structures do not have to be sent by a single descriptor, but can be split up. The device will look at the size of each buffer associated with a descriptor, and reassemble the structure from there. This allows a structure to contain both read-only and write-only fields, as they can be split into descriptors that specify if the device can read or write. Structures can also contain arrays of undefined length, usually for large data transfers. These arrays do not need to be contiguous with the rest of the structure as long as they are referred to by a separate descriptor [21, Chapter 2.6.4].

3.11.2 VIRTIO library

Because VIRTIO uses a common communication protocol for all devices, I implemented a library which handles this communication, to make each driver simpler. The library provides flexible interrupt handling by letting response handlers be set per message, in addition to setting a default response handler for each VIRTIO queue. It uses the platform-specific header files and the interrupt listener functionality described in section 3.8.

The library has the following interface:

- `void virtio_init(void);`
  This function is called during system initialization, and it checks that VIRTIO is available, and collects an internal list of the available devices.

- `virtio_dev *virtio_get_device(int type);`
  This function returns the first unused device of the given type, which corresponds to the `Device ID` in the VIRTIO specification.

- `int virtio_setup(virtio_dev *dev, char *name, virtq_dev_specific_init virtq_init, le64 features);`
  This function resets, configures, and initialized a VIRTIO device. `virtq_init` is called at right time in the negotiation process to allocate the queues needed for the device. `features` is the feature flags the driver supports. Only the features which both the device and driver supports are enabled.

- `void virtio_disable(virtio_dev *dev);`
  This function resets a VIRTIO device.

- `void virtio_enable_interrupt(virtio_dev *dev, virtio_config_change_handler config_change_handler);`
  This function enables interrupts for a VIRTIO device. `config_change_handler` is the listener for device configuration changes.

- `void virtio_disable_interrupt(virtio_dev *dev);`
  This function disables interrupts for a VIRTIO device.

- `int virtq_alloc(virtio_dev *dev, uint queueIdx, ulong size);`
  This function allocates a VIRTIO queue with a given index and size for a VIRTIO device.

- `int virtq_add_desc_chain(virtq *queue, virtq_intr_handler handler, void *handler_data, uint num, ...);`
  This function adds a descriptor chain to the given VIRTIO queue. `handler` is the response handler for the chain. `handler_data` is a value that will be passed to the handler. `num` is the number of descriptors in the chain. For each descriptor there should be three sequential arguments, the address, the size, and a flag indicating whether the descriptor is writable by the device.

- `void virtq_free_chain(virtq *queue, virtq_desc *head);`
  This function will free a previously allocated chain, as long as each descriptor was allocated separately.

- `void virtq_make_available(virtq *queue);`
  This function will make all current descriptor chains in a queue visible to the device.

- `void virtq_notify(virtio_dev *dev, int queuenum, int notify_response, int avail_idx);`
  This function will send a notification to the device of the descriptors in the available ring up to index `avail_id`.

- `virtq_used_elem *virtq_get_next_used(virtq *queue);`
  This function returns the next element in the used ring.

3.11.3 GPU

The VIRTIO GPU device uses one or multiple framebuffers to transmit display data from the driver to the device. The device has a copy of the framebuffer, called a resource, in its own memory. To set up a framebuffer the driver has to request the device to create a resource, then allocate the framebuffer and request that the framebuffer is connected to the resource, then request the device to use the resource for a given scanout (screen). When the screen should be updated the driver must send a message that a region of the framebuffer is invalidated, then request that the device flushes the region of the resource to the screen [21, Chapter 5.7].

At first, I implemented the driver to invalidate and flush the framebuffer for every write. However, this resulted in a lot of small updates, which were visibly slow. Instead, I implemented an update queue, and a timer which drains the queue and flushes each region. Updates which are close together are merged, to reduce the number of messages sent to the device.

3.11.4 Input

The VIRTIO input device represents all kinds of input devices, like keyboards, mice, joysticks etc. Unlike other kinds of devices the input device does not need to be polled, but writes to the next available descriptor whenever an event occurs. The driver allocates all descriptors during initialization, but does not deallocate them after use because they will simply be overwritten the next time the device gets to that index in the descriptor table.

Each input event consists of a type, a code, and a value, conforming to the evdev interface used by the Linux kernel [21, Chapter 5.8]. The evdev interface is described in Torvalds [20, Version 5.12.10, Documentation/input/event-codes.rst]. A full list of the key codes is available in Torvalds [20, Version 5.12.10, include/uapi/linux/input-event-codes.h].

Keyboard drivers in Inferno only interact with the rest of the operating system by adding the typed characters to the keyboard queue `kbdq`. This means that each driver has to keep track of modifier keys, and has to define the keymap. The pc port includes a keyboard driver which supports evdev events, so I used that driver with small modifications to work with VIRTIO. This driver uses the standard US keymap.

I started to implement a mouse driver, which uses the same device type as the keyboard but sends different events and key codes. However, because the window manager is not available (see section 3.14) there is limited use for it, and it is harder to test.

3.11.5 Block device

The VIRTIO block device represents a hard drive, which is usually backed by a file in the host file system. The device is fairly straight-forward to use, the metadata like block size and capacity is given in the device configuration space. Read and write requests are sent on the same format, containing a sector number to start from and an array of data to read from or write to, depending on the operation. The device responds by writing a status code to the end of the request structure.

In Inferno storage device drivers are represented by a `SDifc` structure which contains the name and function pointers to the standard storage device functions, or `nil` if the function is not defined for that device.

I implemented the following storage device functions for this driver:

- `SDev* pnp(void)`
  This function discovers, sets up, and returns a linked list of all storage devices on this interface.

- `SDev* id(SDev*)`
  This function gives each storage device in the given linked list a unique name. I used the naming scheme "virtblkX", where X is an incrementing number.

- `int enable(SDev*)`
  This function enables interrupts from the given device. Returns 1 if successful, otherwise returns 0.

- `int disable(SDev*)`
  This function disables interrupts from the given device. Returns 1 if successful, otherwise returns 0.

- `int verify(SDunit*)`
  This function performs the equivalent of an SCSI inquiry command. Returns 1 if successful, otherwise returns 0.

- `int online(SDunit*)`
  This function retrieves the storage device block size and storage capacity. Return 1 if successful, otherwise returns 0.

- `long bio(SDunit* unit, int lun, int write, void* data, long nb, long bno)`
  This function performs a read or write request to or from the buffer `data`, starting at block `bno` until block `bno+nb`. Returns the number of bytes read or written. Because the function can not return the number of bytes until the operation is finished, the function is blocking. The function name probably means "buffered I/O", as it is linked to the Limbo library Bufio [11].

It is worth mentioning that there is an alternative to `bio` in the `int rio(SDreq*)` function. I decided not to implement this yet because `SDreq` seems to be based on SCSI, and `bio` seemed much easier to implement. From reading other implementations of `rio` I am not sure how it is supposed to work, but the name might mean "raw I/O".

The full driver implementation is included in appendix D.

3.12 Graphical output

In addition to the GPU driver there has to be an interface between Inferno and the driver which implements screen functions used in Inferno. For this I used the `screen.h` and `screen.c` files from Richard Miller's port of Plan9 to Raspberry Pi, modified for Inferno by *Lab 18, we have a screen!* [13]. This interface is designed for a framebuffer, so it was easily adapted to the VIRTIO GPU driver.

With these files in place a border is drawn around the screen when Inferno starts. All printed text, expect that printed only to UART with `iprint`, is displayed on the screen. User input is printed as the user types in it. When the text reaches the bottom the window is scrolled down, to keep the most recent text in view.

3.13 Initializing the system

When QEMU starts it first gives control to OpenSBI, running in machine mode. OpenSBI sets up the machine, then calls the kernel in supervisor mode at address `0x80400000`. The function at that address is called `_start()`, which is shown in listing 4.

The Plan9 assembler usually inserts a function prologue which allocates `x+4` bytes stack space automatically, based on the `$x` parameter, and stores the link register. An epilogue is inserted to load the link register and reset the stack. However, because the stack pointer is not initialized yet the first function has to be declared with `$-4`, which prevents the assembler from inserting a prologue and epilogue.

The `_start()` function sets up the registers for the rest of the kernel. It sets the stack pointer, register `R2`, to a predefined address from `mem.h`. It also uses a pseudoinstruction to set the static base, which is the address at the start of the kernel, to register `R3` for relative addressing. After the registers are initialized it calls `main()`, which continues the initialization from C code.

```
#include "mem.h"

TEXT _start(SB), $-4
    /* set static base */
    MOVW $setSB(SB), R3

    /* set stack pointer */
    MOVW $(MACHADDR+MACHSIZE-4), R2

    /* call main */
    JAL R1, main(SB)
```

Listing 4: The `_start()` function

The `main()` function first initializes the memory for the kernel [12]. First the bss section, used for static variables and located after the kernel binary, is cleared. Then the memory pool, located after the bss section until the end of memory, has to be defined for the kernel to know which portions it can use. Currently, the size of the memory is not checked at runtime, so the emulator has to be started with at least the same amount of memory as the kernel expects, which is currently 128 MiB.

After the memory is initialized, traps are enabled and timers are initialized. Then the print queue and device drivers, like UART, input, and GPU, are initialized. Then the screen is initialized, and the OS information is printed.

Finally, user processes and the VM are initialized, and the Dis binary `/osinit.dis` is executed.

The `main()` function is shown in listing 5. The full `main.c` file is included in appendix C.

3.14 Interactive shell

Starting the interactive shell is the baseline for a usable Inferno installation. For the shell to be available, the Dis file for the shell itself and all programs which should be available from the shell must be included in the `root` section of the platform configuration file. The shell is started from the Dis init file, by loading the shell module and spawning a shell instance in a new thread. If shell commands should be executed during initialization, they can be executed directly using the shell module. The init code necessary to start the shell is included in listing 6. A screenshot of the system after starting the shell and running the `ls` command is shown in figure 1.

```c
void
main() {
    // Clear bss
    memset(edata, 0, end-edata);
    memset(m, 0, sizeof(Mach));

    // Initialize the memory pool
    confinit();
    xinit();
    poolinit();
    poolsizeinit();

    // Enable traps and timers
    trapinit();
    clockinit();

    // Set up UART and the print queue
    printinit();
    i8250console();

    // Set up VIRTIO drivers
    virtio_init();
    input_init();

    // Initialize the screen
    screeninit();

    print("\nRISC-V QEMU\n");
    print("Inferno OS %s Vita Nuova\n\n", VERSION);

    // Start processes
    procinit();
    links();
    chandevreset();

    eve = strdup("inferno");

    userinit();
    schedinit();
}
```

Listing 5: The `main()` function

The next step up from the interactive shell is to start the window manager. However, the wm requires so many smaller programs to be included that it is unsuited to be compiled in the binary, and should be provided using a harddrive. However, as will be discussed in section 3.15, this is not possible yet.

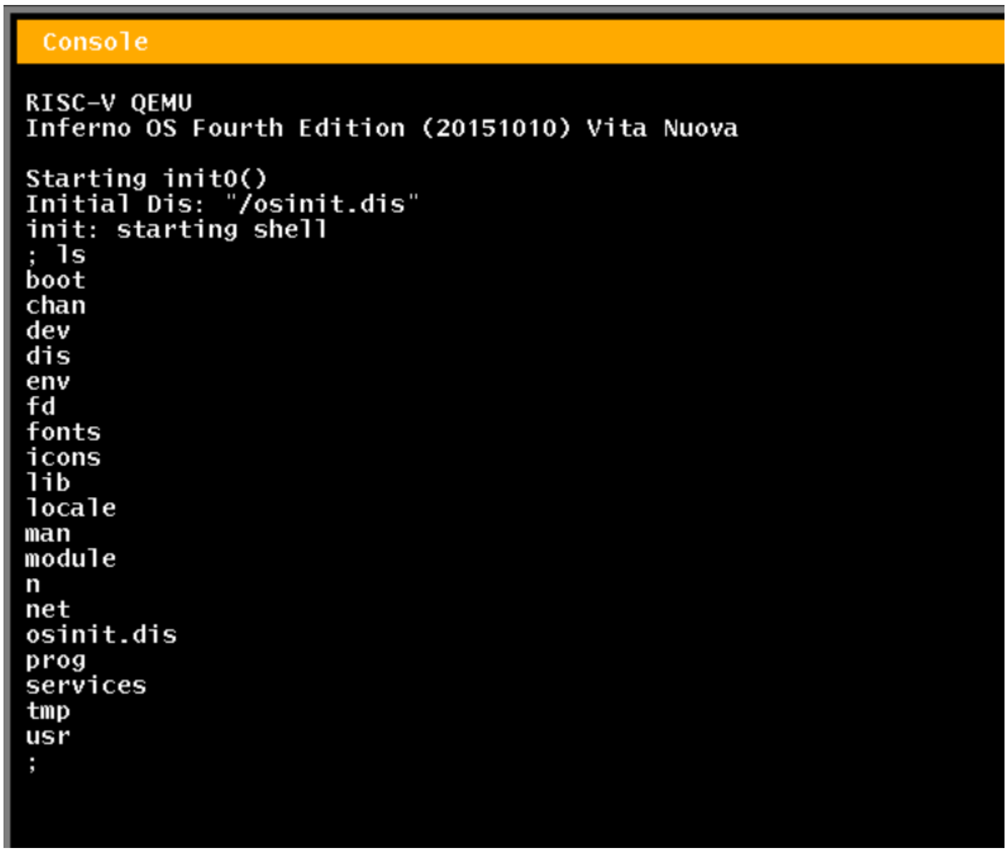
```
implement Init;

include "sys.m";
        sys: Sys;
        print: import sys;
include "sh.m";
        sh: Sh;
include "draw.m";
        draw: Draw;
        Context: import draw;

Bootpreadlen: con 128;

Init: module
{
    init:   fn();
};

init()
{
    sys = load Sys Sys->PATH;
    sh = load Sh Sh->PATH;

    sys->bind("#i", "/dev", sys->MREPL);   # draw device
    sys->bind("#c", "/dev", sys->MAFTER);  # console device
    sys->bind("#S", "/dev", sys->MAFTER);  # storage devices

    spawn sh->init(nil, "sh" :: "-i" :: nil);
}
```

Listing 6: The Limbo code to start the interactive shell

3.15 Filesystem

As mentioned in 3.4.3, a simple filesystem is included in the binary to provide the programs needed to initialize the system. However, this filesystem is read-only, and while it is possible to cram in all the available Limbo programs, the binary quickly becomes unreasonably large. Instead, a separate filesystem should be mounted to provide the rest of the Limbo programs, and user-modifiable files. This could be done over the network to another computer using the 9P protocol, but for this project I used a harddrive utilizing the VIRTIO block device driver, as described in 3.11.5.

The harddrive QEMU presents to the driver is backed by a file in the host filesystem, where I allocated a partition usable for Inferno. First I tried using the kfs filesystem native to Plan9 and



```
Console

RISC-V QEMU
Inferno OS Fourth Edition (20151010) Vita Nuova

Starting init0()
Initial Dis: "/osinit.dis"
init: starting shell
; ls
boot
chan
dev
dis
env
fd
fonts
icons
lib
locale
man
module
n
net
osinit.dis
prog
services
tmp
usr
;
```

Figure 1: The system after starting the shell and running a command.

Inferno, but I had trouble finding Linux tools for it on the host side. It is possible to run Inferno hosted under Linux to format the partition, however that was a very cumbersome process. In addition, when mounting the filesystem in Inferno running on RISC-V, the kfs driver constantly had to check the filesystem, and froze when trying to mount it.

Instead, I used the FAT filesystem, which Linux fully supports. Inferno has a driver for FAT32, however it is uncertain how well all the features and extensions of the filesystem is supported.

After creating and partitioning the harddrive file, I mounted it on the host system and copied over the entire `/dis/` folder with all the compiled Limbo programs. I then added the command line parameters shown in listing 7 to QEMU to use the file as the harddrive, accessible as a VIRTIO block device. The drive is detected by the driver at boot, and is available in Inferno under `/dev/virtblk00/`. However, the partitions are not detected or represented by files automatically. To do that, the `fdisk` tool has to read the partition table and write the configuration to the disk control file. Because the partition type is FAT, the partition file will automatically be `/dev/virtblk00/dos`. Then the partition file must be mounted using the `dossrv` tool. Finally, the `dis` folder on the harddrive has to be bound to `/dis`, so all the program files are where they are expected. The full commands to achieve this is listed in listing 8. To reduce the number of manual commands during system setup, these commands are executed in the init file, before the shell is started.

Unfortunately the filesystem is read very slowly, because the block device driver is asked to read small sequential blocks. In addition, the system freezes halfway through reading files from the filesystem, like when using the kfs filesystem. However, it is unclear if this bug is in the block device driver, the filesystem driver, or some other program.

```
-drive if=none,format=raw,file=hdd.img,id=hdd -device
 ↪  virtio-blk-device,scsi=off,drive=hdd
```

Listing 7: The flags passed to QEMU to set up the hard drive with the VIRTIO block device driver.

```
disk/fdisk -p /dev/virtblk00/data > /dev/virtblk00/ctl
dossrv -f /dev/virtblk00/dos -m /n/local
bind /n/local/dis /dis
```

Listing 8: The Inferno shell commands to set up and mount the filesystem on a harddrive

3.16 The Just-in-time compiler

A Just-in-time (JIT) compiler dynamically translates one set of instructions to instructions native to the processor it is running on, at runtime [2]. This approach sacrifices some time and memory to compile the program, but the result will run faster than when using an interpreter. How fast the JIT compiles, and how fast the resulting code runs, depends on the similarity between the instruction sets, and which optimizations the JIT performs.

The Inferno OS includes a framework for JIT compilers which compile Dis programs to native instructions. As all user space programs are Dis programs in Inferno, a JIT compiler is essential to get a responsive system.

3.16.1 The Dis instruction set

The Dis virtual machine uses an instruction set modeled after CISC-processors, providing three-operand memory-to-memory instructions. The authors compare this approach to that of the Java stack-based virtual machine, and notes that the memory-to-memory approach is closer to common processors and makes the JIT compiler more efficient on non stack-based processors [26].

The instructions are organized into modules, which are loaded and compiled to native code separately. Each module has a data segment, and each function gets allocated a frame for local variables. The instructions can access values in the module data or function frame, or indirectly access values whose addresses are stored in one of those locations.

The instruction set has instructions for various datatypes, including 8-bit unsigned integers, 32 and 64-bit signed integers, 64-bit double precision floating-point, UTF-8 encoded strings, pointers, memory, and memory containing pointers.

The virtual machine uses reference-counted garbage collection [26]. As a result of this, pointers have to be handled using special instructions, to ensure that the garbage collector tracks every instance of the pointer. This includes instructions which allocate memory, so that task is moved from the programmer to the virtual machine [7].

The virtual machine has a few registers, to store the program counter, module data pointer, function frame pointer etc., but the registers are not directly accessible through the instruction set.

3.16.2 The structure of the virtual machine

The Dis virtual machine defines a C struct for the virtual registers, which is used by the interpreter and the compiled instructions. When moving between the interpreter and compiled code, or from the interpreter to an instruction handler, the normal C calling convention is disregarded, and the virtual registers are used instead. The handler for each instruction is separate from the rest of the interpreter, so the compiled code can call a handler in isolation if there is an instruction that is too complex or too infrequent to implement in the JIT compiler.

When reaching the entry point of each module the virtual machine will check whether the module has been compiled yet, execute it if it has, or try to compile it if it has not. If the compilation fails it uses the interpreter as a fallback. It seems to be possible to set the `MUSTCOMPILE` or `DONTCOMPILE` flags in the Dis binary to either force the module to be compiled, or be handled by the interpreter [6]. However, it does not seem like these flags are used by the Limbo compiler.

The Inferno JIT compilers are very simplistic compilers. They use a mixed code approach [2], but the decision to use native or interpreted code is done per instruction based on complexity, not based on how frequently a section is executed. After the first compilation, Inferno does not call the JIT compiler again for the same module, so no further optimizations are possible.

3.16.3 The structure of the JIT compilers

The only public function of a JIT compiler is the `compile` function, which is called when a new module should be compiled. However, the existing JIT compilers seem to follow the same basic structure.

The compilation is done in two passes. In the first pass each compiled Dis instruction is overwritten by the next one so that the total size and offsets are known for the second pass.

The JIT compiler will try to optimize the compiled instructions based on the information in the instruction, like in the size of the datatype, or by calculating based on the immediate value.

There are many Dis instructions which are complex or not supported by the native instruction-set. These are often delegated to the interpreter by loading the operators into the virtual registers and calling the handler function for that instruction.

The JIT compilers often add macros, which are basically functions, to reduce code duplication of sections which are general and is not optimized at compile time. These macros are placed after each module.

3.16.4 Implementing the JIT for RISC-V

The best way to implement a new JIT compiler would be to start by delegating all instructions to the interpreter, then implementing one instruction at a time. However, I did not understand the way the Dis JIT compilers usually worked when I started this, so I did not see that possibility. Instead, I decided to look through the code of another JIT compiler line by line, and copy or translate each line as I understood what it did. This means that my JIT compiler implements roughly the same instructions as the one I based the code on. I mostly based my code on the ARM JIT compiler, because I am most familiar with ARM assembly. However, while ARM and RISC-V are RISC architectures, their instruction sets are quite different, so translating was sometimes hard. I sometimes used the MIPS JIT compiler as a second reference because the instruction set it much closer to RISC-V, but the structure and naming convention made the code hard to read.

The JIT compiler has to be careful how it allocates registers. RISC-V usually has 32 registers, however the Plan9 compiler only uses the first 16 to be more compatible with compressed instructions and the planned RV32E instruction set, which only has 16 registers. I decided to use the same restriction for the JIT compiler. Three registers have to be permanently reserved for the current frame pointer, module pointer, and pointer to the virtual registers, five registers are used for storing values for a single Dis instruction, one register is used for constructing 32-bit numbers from immediate values, and one register is used to store the address when loading double indirect operands. Finally, one register is used to store `H`, the Dis value for invalid pointers. Keeping `H` in a register simplifies comparing values to `H`, since it otherwise would have to be loaded into a register each time.

The JIT compiler starts by compiling a module preamble, which sets up the fixed registers, then jumps to the first compiled instruction. Then the first pass is compiled, each instruction overwriting the last, storing the compiled size of each Dis instruction. Then the buffer for the second pass is allocated based on the sizes, and the second pass then writes into the allocated buffer. Finally, initializers and destructors for each datatype is compiled.

The current implementation of the JIT compiler assumes that the **M** and **D** RISC-V extensions are supported by the processor. Arithmetic operations on 64-bit integers are emulated using 32-bit integers instead. Arithmetic operations for 32- and 64-bit integers and 64-bit floating-point numbers is implemented, and have undergone some simple test cases.

The conversion between 64-bit integers and 64-bit floats was initially delegated to the interpreter, however this uncovered an issue with the C compiler. When casting a float to an integer in C, the compiler will round the value by adding 0.5 or -0.5, depending on the sign, and then convert it in software, rounding down to the closest integer. However, the C compiler seems to expect that some registers hold float constants, like 0.5, but because I did not know about this these registers have not been set up, and the registers default to `NaN`. For the JIT compiler I stepped around these problems by handling more of the conversion in assembly. For conversions from floating-point to 64-bit integer I handle the rounding in assembly, then call the `_d2v` function, which uses bit manipulation to handle the rest of the conversion. For conversions from 64-bit integers to floating-point I translated the algorithm in the `_v2d` function to assembly, then optimized it to eliminate branches and reduce the number of instructions. The result is that these operations have been implemented using short and efficient assembly code, and the problem with the C compiler has been circumvented.

When writing the code I added comments to explain the logic and exactly what was happening in the generated instructions, both to make it easier for me to come back to, and for future readers looking to understand the JIT compilers.

The full implementation of the RISC-V JIT compiler is included in appendix E.

3.16.5 Testing the JIT

The JIT compiler can be enabled by setting the `cflag` global variable in the configuration file higher than 0. The virtual machine will then try to compile all Dis modules before executing them. Of course, in a 2.5k line JIT compiler implemented in one go there was bound to be bugs. The first test run crashed with an illegal access exception, so I started working on ways to ease the debugging process, to more easily fix this and future bugs.

The most important debugging tool for the JIT compiler is the disassembler. While not required for the JIT compiler to work, it is common to implement a disassembler a separate file. For RISC-V I implemented the disassembler in the file `/libinterp/das-riscv.c`. The disassembler simply takes a pointer to the start of the compiled instructions, and the number of instructions, and prints out the address and assembly for each instruction. I used the standard RISC-V assembly syntax instead of the Plan9 assembly syntax, because it closely resembles how the instructions are laid down in the JIT compiler code. The JIT compiler calls the disassembler for each instruction after pass 2, prefaced with information about the associated Dis instruction. This makes it easy to follow the flow of the program, and check that the fields of the Dis instruction were used correctly.

Sometimes it can be useful to isolate the compiled code for a Dis instruction, to verify that it is correct despite other Dis instruction implementations which have bugs. In such cases it is useful to make all other Dis instructions use the interpreter handler during testing.

One easy way to check the logic of the compiled program is to insert an illegal instruction. This causes an exception, and the exception handler prints out the contents of the registers. For this I created the macro `CRASH()` which inserts a 32-bit 0, which is an illegal instruction in RISC-V. I have used this to check values loaded from memory, and to check which path of a branch was taken.

Sometimes the compiled code calls to C code, which crashes on illegal pointers. The stack trace will often track back to the calling compiled code, but not further because the compiled code does not use the stack. In these cases the called C code can be modified to print the contents of the virtual registers, which contains the PC of the associated Dis instruction, and arguments to the called function.

There have been many bugs in the JIT compiler, including illegal memory accesses, incorrect jump and branch offsets, mistranslations from ARM assembly, etc. Currently, the JIT compiler can correctly compile a simple print and if-statements. However, when trying to start the full shell the program crashes because of an index error. Further work is required to fix all the bugs and make the JIT compiler fully usable.

4 Roadblocks

4.1 Debugging

Debugging is an important part of any software development process, and even more so with something as complex as an operating system. However, operating systems are harder to debug because they lack the inherent framework that applications running inside operating systems have. Especially in the early stages of development it is hard to get any debugging information out from the system.

Normal print-debugging can not be used until the system has established a communication channel with the outside, normally through UART, or SBI if the bootloader supports it. It is possible to print to the screen once that is set up, but the screen updates slower, and might not update at all during crashes. In addition, because the graphical pipeline is more complicated than the serial pipeline, trying to print to the screen can trigger bugs and even crash or freeze the system. Therefore, all debug printing is done through UART, while all other prints go both through UART and to the screen.

One alternative is GDB. Normally GDB is a very useful tool, and it is supported by QEMU, so it can debug from the very first instruction. However, most of the functionality of GDB is not available when running Inferno because of incompatibility with the Plan9 compilers. Plan9 and Inferno have their own symbol table format, which GDB does not support, and even if it could, the symbol table does not include enough information about types and line numbers to be usable. The result is that GDB can be used to debug, but it can not link the instructions in the binary to the source code. It can only be used to show a disassembly at the current location, show the value of registers, and set breakpoints at addresses. Passing the `-a` flag to the linker causes it to print the Plan9 assembly for the whole binary, including addresses, which can be used to figure out where to set breakpoints. However, because Plan9 assembly is different from the style used in GDB, and it includes many pseudoinstruction, figuring out which part is currently executing and finding bugs is very hard, and requires a lot of cross-referencing.

Inferno includes a debugger, called Acid [24, 9], which can debug the kernel [25], and which can be used from the host system through a serial port connection. To do this with QEMU, the serial port has to be exposed as a virtual `tty` using the `-serial pty` flag to QEMU. Then Acid has to be executed on the host system, with the command `acid -R <pty path> <OS binary path>`. However, the OS does not seem to respond to the messages sent by Acid, instead interpreting them as normal user input. This might be because the UART driver or some other step in the pipeline does not correctly identify the Acid control sequence.

The current debugging situation is not ideal, and it means that a lot of time is required to debug even the smallest bugs. The best solution would be to make the linker output contain enough debugging information for GDB to use, but that would require major alterations to the code and structure of the compiler and linker. With UART working, debugging by printing has become a simpler alternative for most situations, at least to find where the bug is.

When debugging the JIT compiler, the debugging methods I use for C does not work as well. The method I have used the most is to insert an illegal instruction as a breakpoint, which causes the trap handler to print information about the registers and the stack. While less flexible than other methods, it is very easy and fast to add and remove the illegal instructions, and to recompile and test. I sometimes use GDB, but that requires setting a breakpoint after each module is compiled, then looking through the disassembled instructions generated by the JIT and find the right address to break on. This makes it a lot more cumbersome to do rapid incremental testing. Inserting debug print statements into the compiled instruction stream is technically possible, but requires a lot of extra code and setup to make efficiently usable in assembly. However, because the compiled code often calls the interpreter, print statements can be inserted into the C code of the interpreter.

4.2 Lack of documentation

While Inferno (or rather Plan9) does have good documentation in general, the documentaion relevant to porting, such as documentation of the kernel functions, compilers, and utilities, are at best a minimal description of behavior, not a full guide.

For example, the documentation for the assembler explains addressing modes, function definitions, and calling conventions, but does not list the available instructions. Because Plan9/Inferno uses an assembly syntax that is meant to be similar for all platforms, but therefore is very different from more common assembly dialects, it is very hard to be sure what the assembly code actually does. I have had to read through the lexer and parser code on several occasions to figure out what an instruction does or how it should be used.

In addition, there is basically no official documentation for porting Inferno to new architectures or platforms. That means that I usually have to read through the code for other platforms, and develop incrementally, figuring out what I need to implement based on the errors I receive. This can be quite difficult because the compiler has vague error messages, and sometimes does not refer to the error location.

The result of this is that this project was very slow and time-consuming compared to how much code I had to write.

4.3 Floating-point problems

As mentioned in section 3.16.4, I came across some problems with the Plan9 C-compiler when performing floating-point arithmetic, because it assumes that certain registers hold common double-precision constants. While I discovered this while working on the JIT compiler, this problem affects all double-precision floating-point operations that uses the values the compiler assume are in registers.

Looking through the source code, it seems like this approach is fairly common, though it is not present in the x86 and ARM compilers. The registers the RISC-V compiler expects are listed in table 1. Floating-point constants that are not in this list are either constructed from other constants, or put into the data segment by the linker, and loaded when needed.

To solve this I added a function that is called during system initialization, which loads these values into the registers. With this change, double-precision floating point operations work as expected in C and when using the interpreter.

Because double-precision values can not be used for single-precision operations, and vice versa, separate constants are needed for single-precision operations. Because double-precision is the primary Dis floating-point type, and single-precision is only included for compatibility [7], single-precision values are not given permanent registers, and are always loaded from memory.

However, there seems to be a problem with the single-precision constants. All single-precision constants I have tested have had the hexadecimal representation `0x000001f4`. After digging around in the compiler code I discovered that when the linker writes the constant to the data segment, it is read from memory as if it was the highest four bytes of a double-precision value. However, because the value is stored in a four byte single-precision value, the read overflows. See listing 9. This bug is not present in the other compilers, and once found it was easy to fix.

Table 1: The floating-point constant register the compiler expects.

FP register	Value
28	0.0
29	0.5
30	1.0
31	2.0

```
fl = ieeedtof(p->to.ieee);
cast = (char*)&fl;
for(; i<c; i++) {
    buf.dbuf[l] = cast[fnuxi8[i+4]];  // Original version
    buf.dbuf[l] = cast[fnuxi4[i]];    // Fixed version
    l++;
}
```

Listing 9: The bugged code from `utils/il/asm.c` that writes single-precision floating-point constants to the data section.

4.4 Random crashes

The OS seems to have several bugs that trigger at random, which are hard to track down and fix. Because these issues usually appear right after booting, and less than half the time, I have not made them a priority to fix, and instead just try to start the OS again. However, they are serious problems that have to be fixed before Inferno on RISC-V can be used for any real-world purpose.

Here is a list of the issues that I know of:

- There is a problem with the memory management functions that crashes the system. The likelihood of it triggering seems to scale with the number of allocations that are requested. From my testing, the problem seems to be a buffer overflow which overwrites the heap metadata, causing a crash when the memory management functions walks the heap. The source of this overflow will be hard to determine.
- Once in a while, right after starting, QEMU reports an illegal memory access and crashes, before the OS got far enough to print. It might even be a problem with OpenSBI. This should be easy to debug with GDB, however it rarely happens, so one has to set up GDB and run QEMU again and again until it happens. Print-debugging would have to be done through SBI as the crash happens before the UART driver is set up.
- Very rarely the system seems to freeze at boot, without printing anything and without any error messages from QEMU. This might be because of a loop that does not end properly, but it's hard to determine because of the same difficulties with debugging as the issue above.

4.5 The timer bug

As mentioned in section 3.9.3, setting timer interrupts infinitely far in the future did not work as expected. While this does not affect programs which sets new timers continuously, like an operating system scheduling processes, it can cause problems for programs which only occasionally set timers.

I decided to test this by writing a small program that only set the timer, to reduce the number of possible errors. I discovered that this problem exists for both 32-bit and 64-bit RISC-V, and both when running in **S** mode and setting timers through OpenSBI, and when running in **M** mode and setting the timers directly. This means that the problem lies in QEMU. After looking through the QEMU source code I found the function `sifive_clint_write_timecmp` in the file `/hw/intc/sifive_clint.c`, which handles setting timers for all RISC-V platforms in QEMU. The function is listed in listing 10.

The first issue is on line 63, which converts the number of ticks until the next timer to nanoseconds. For the virtual RISC-V target, `NANOSECONDS_PER_SECOND/timebase_freq` is 100, which is then multiplied with `diff`, causing an overflow. This is not really a problem if the timer is set to -1, because the result is still close to -1, but if only the most significant bit is set it will overflow to zero, causing an immediate interrupt.

The second problem is that `timer_mod`, which is the general function for setting timers for all QEMU platforms, takes a signed 64-bit integer as its second argument. A bit further down the chain, in `timer_mod_ns_locked`, this value is set to 0 if it was below 0.

These two problems combined mean that if the number of ticks until the next timer, multiplied with 100, has the most significant bit set, a timer interrupt is triggered immediately.

This bug has been reported to the QEMU developers.

```c
/*
 * Called when timecmp is written to update the QEMU timer or immediately
 * trigger timer interrupt if mtimecmp <= current timer value.
 */
static void sifive_clint_write_timecmp(RISCVCPU *cpu, uint64_t value,
                                       uint32_t timebase_freq)
{
    uint64_t next;
    uint64_t diff;

    uint64_t rtc_r = cpu_riscv_read_rtc(timebase_freq);

    cpu->env.timecmp = value;
    if (cpu->env.timecmp <= rtc_r) {
        /* if we're setting an MTIMECMP value in the "past",
           immediately raise the timer interrupt */
        riscv_cpu_update_mip(cpu, MIP_MTIP, BOOL_TO_MASK(1));
        return;
    }

    /* otherwise, set up the future timer interrupt */
    riscv_cpu_update_mip(cpu, MIP_MTIP, BOOL_TO_MASK(0));
    diff = cpu->env.timecmp - rtc_r;
    /* back to ns (note args switched in muldiv64) */
    next = qemu_clock_get_ns(QEMU_CLOCK_VIRTUAL) +
        muldiv64(diff, NANOSECONDS_PER_SECOND, timebase_freq);
    timer_mod(cpu->env.timer, next);
}
```

Listing 10: The implementation of the timer handling for RISC-V in QEMU.

5 Conclusion

Through this project I have investigated and tried to solve the challenges of porting the Inferno operating system to RISC-V, including setting up the boot process, programming trap handling, some important drivers, and a Just-In-Time compiler. I have not investigated the challenges of actual hardware platforms, but I believe that porting Inferno to such devices, with the necessary capabilities, should be easy after the groundwork I have done here.

At the end of this project I have created a port of Inferno to RISC-V which can print and receive input, output to a screen, run user processes in an interpreter, and mount a harddrive. In addition, I have started the work on a JIT compiler, which can compile arithmetic and function call instructions correctly.

6 Future work

There is still a lot of work to do for Inferno to be fully usable on RISC-V. The crashes mentioned in section 4.4 have to be fixed, the block-device driver has to be improved to increase the performance, and prevent the freeze mentioned in section 3.15. More drivers have to be implemented, especially for real hardware. In addition, the JIT compiler has to be fully tested and fixed to increase the performance of the system. Multi-core support can also be added to increase performance.

After these issues are fixed, and the system has been ported to real hardware, it can be tested on a network of embedded devices. This will show whether Inferno on RISC-V is practical and competitive for the IoT market.

7 Acknowledgments

I would like to thank my supervisor, Michael Engel, for his help and advice through this project.

I would also like to thank Richard Miller, who wrote the Plan9 RISC-V compiler, without which this project would not have gotten as far.

8 Bibliography

[1] Krste Asanović and David A Patterson. “Instruction sets should be free: The case for RISC-V”. In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014).

[2] John Aycock. “A brief history of just-in-time”. In: *ACM Computing Surveys (CSUR)* 35.2 (2003), pp. 97–113.

[3] Fabrice Bellard. “QEMU, a fast and portable dynamic translator.” In: *USENIX Annual Technical Conference, FREENIX Track.* Vol. 41. 2005, p. 46.

[4] Palmer Dabbelt, Drew Barbier, and Abner Chang, eds. *RISC-V Platform-Level Interrupt Controller Specification.* RISC-V Foundation. Mar. 2020. URL: `https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic.adoc` (visited on 12/08/2020).

[5] Palmer Dabbelt and Atish Patra, eds. *RISC-V Supervisor Binary Interface Specification.* RISC-V Foundation. Sept. 2020. URL: `https://github.com/riscv/riscv-sbi-doc/blob/master/riscv-sbi.adoc` (visited on 12/08/2020).

[6] *Dis object file.* Vita Nuova Limited. URL: `http://www.vitanuova.com/inferno/man/6/dis.html` (visited on 07/18/2021).

[7] *Dis Virtual Machine Specification.* Lucent Technologies Inc, Vita Nuova Limited. Sept. 2000. URL: `http://www.vitanuova.com/inferno/papers/asm.pdf` (visited on 06/05/2021).

[8] Sean Dorward et al. *The Inferno Operating System.* Lucent Technologies, Bell Labs. 1997. URL: `http://www.vitanuova.com/inferno/papers/bltj.pdf` (visited on 12/08/2020).

[9] Tad Hunt. *Acid Reference Manual.* Vita Nuova, Lucent Technologies. 2000. URL: `http://www.vitanuova.com/inferno/papers/acid.pdf` (visited on 08/01/2021).

[10] *Inferno OS.* 2014. URL: `http://lynxline.com/category/inferno/` (visited on 12/08/2020).

[11] Brian W. Kernighan. *A Descent into Limbo.* Vita Nuova, Bell Labs. 2005. URL: `http://doc.cat-v.org/inferno/4th_edition/limbo_language/descent` (visited on 08/01/2021).

[12] *Lab 10, Bss, memory pools, malloc.* 2013. URL: `http://lynxline.com/lab-10-bss-menpools-malloc/` (visited on 12/08/2020).

[13] *Lab 18, we have a screen!* 2013. URL: `http://lynxline.com/lab-18-we-have-a-screen/` (visited on 06/15/2021).

[14] *Lab 9, coding assembler part.* 2013. URL: `http://lynxline.com/lab-9-coding-assembler-part/` (visited on 12/08/2020).

[15] Richard Miller. Nov. 9, 2020. URL: `https://bitbucket.org/inferno-os/inferno-os/pull-requests/8` (visited on 07/30/2021).

[16] Richard Miller. Mar. 3, 2021. URL: `https://9p.io/sources/contrib/miller/riscv-old.tar` (visited on 07/30/2021).

[17] Richard Miller. May 28, 2021. URL: `https://9p.io/sources/contrib/miller/riscv.tar` (visited on 07/30/2021).

[18] Rob Pike et al. *Plan 9 from Bell Labs.* Bell Labs. URL: `http://9p.io/sys/doc/9.html` (visited on 12/08/2020).

[19] *QEMU.* URL: `https://gitlab.com/qemu-project/qemu` (visited on 07/30/2021).

[20] Linus Torvalds, ed. *Linux Kernel.* URL: `kernel.org` (visited on 06/05/2021).

[21] *Virtual I/O Device (VIRTIO) Version 1.1.* Oasis Open. Apr. 2019. URL: `https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.pdf` (visited on 06/05/2021).

[22] *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2019121.* Tech. rep. RISC-V Foundation, Dec. 2019.

[23] *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified.* Tech. rep. RISC-V Foundation, June 2019.

[24] Phil Winterbottom. *Acid: A Debugger Built From A Language.* 1995. URL: `http://doc.cat-v.org/plan_9/2nd_edition/papers/acid/` (visited on 12/08/2020).

[25] Phil Winterbottom. *Native Kernel Debugging with Acid.* URL: `http://doc.cat-v.org/inferno/4th_edition/kernel_debugging/` (visited on 06/05/2021).

[26] Phil Winterbottom and Rob Pike. *The design of the Inferno virtal machine.* Lucent Technologies, Bell Labs. URL: `http://www.vitanuova.com/inferno/papers/hotchips.pdf` (visited on 06/05/2021).

Appendices

A /os/virtriscv/virtriscv

```
dev
	root
	cons
	env
	mnt
	pipe
	prog
	srv
	dup
	uart
	sd

	pointer
	draw	screen
	pointer

	ip	bootp ip ipv6 ipaux iproute arp netlog ptclbsum iprouter plan9 nullmedium pktmedium netaux

ip
	tcp
	udp
	ipifc
	icmp
	icmp6
	ipmux

lib
	interp
	math
	draw
	memlayer
	memdraw
	tk
	sec
	kern

misc
	uarti8250
	sdvirtblk

mod
	sys
	draw
	tk
	math

port
	alarm
	alloc
	allocb
	chan
	dev
	dial
	dis
	discall
	exception
	exportfs
	inferno
	latin1
	nocache
	nodynld
	parse
	pgrp
	print
	proc
	qio
	qlock
	random
	sysfile
	taslock
	xalloc

code
	int kernel_pool_pcnt = 10;
```

```
        int main_pool_pcnt = 40;
        int heap_pool_pcnt = 20;
        int image_pool_pcnt = 40;
        int cflag=0;
        int swcursor=1;
        int consoleprint=1;

init
        virtriscvinit

root
        /chan   /
        /dev    /
        /dis
        /lib    /
        /env    /
        /fd     /
        /net    /
        /prog   /
        /n      /
        /n/local    /
        /n/dos  /
        /tmp    /
        /dis/lib
        /dis/disk
        /osinit.dis
        /dis/sh.dis
        /dis/tiny/sh.dis
        /dis/ls.dis
        /dis/mc.dis
        /dis/lc
        /dis/ps.dis
        /dis/ns.dis
        /dis/cat.dis
        /dis/bind.dis
        /dis/mount.dis
        /dis/mntgen.dis
        /dis/listen.dis
        /dis/export.dis
        /dis/unmount.dis
        /dis/sleep.dis
        /dis/pwd.dis
        /dis/echo.dis
        /dis/cd.dis
        /dis/netstat.dis
        /dis/styxlisten.dis
        /dis/time.dis
        /dis/lib/arg.dis
        /dis/lib/auth.dis
        /dis/lib/lock.dis
        /dis/lib/rand.dis
        /dis/lib/random.dis
        /dis/lib/dial.dis
        /dis/lib/bufio.dis
        /dis/lib/timers.dis
        /dis/lib/string.dis
        /dis/lib/filepat.dis
        /dis/lib/readdir.dis
        /dis/lib/workdir.dis
        /dis/lib/daytime.dis
        /dis/lib/nametree.dis
        /dis/lib/styxservers.dis
# disk support
        /usr    /
        /usr/inferno    /
        /dis/dd.dis
        /dis/fs.dis
        /dis/dossrv.dis
        /dis/lib/fslib.dis
        /dis/lib/fsproto.dis
        /dis/lib/fsfilter.dis
        /dis/zeros.dis
        /dis/disk
        /dis/disk/calc.tab.dis
        /dis/disk/fdisk.dis
        /dis/disk/format.dis
```

```
	/dis/disk/ftl.dis
	/dis/disk/kfs.dis
	/dis/disk/kfscmd.dis
	/dis/disk/mbr.dis
	/dis/disk/mkext.dis
	/dis/disk/mkfs.dis
	/dis/disk/pedit.dis
	/dis/disk/prep.dis
	/dis/lib/disks.dis
	/dis/lib/styx.dis
# misc
	/dis/math/sieve.dis
# structure
	/boot	/
	/man	/
	/fonts	/
	/icons	/
	/module	/
	/locale	/
	/services	/
```

B /os/virtriscv/mkfile

```
#                                               -*-makefile-*-
<../../mkconfig

#Configurable parameters

CONF=virtriscv                  #default configuration
CONFLIST=virtriscv

SYSTARG=$OSTARG
OBJTYPE=riscv
INSTALLDIR=$ROOT/Inferno/$OBJTYPE/bin   #path of directory where kernel is installed

LOADADDR=0x80400000

<$ROOT/mkfiles/mkfile-$SYSTARG-$OBJTYPE #set vars based on target system

<| $SHELLNAME ../port/mkdevlist $CONF

HFILES=\
        mem.h\
        dat.h\
        fns.h\
        io.h\

OBJ=\
        load.$O\
        clock.$O\
        portclock.$O\
        mul64fract.$O\
        tod.$O\
        plic.$O\
        sbi.$O\
        inb.$O\
        dump.$O\
        csr.$O\
        trap.$O\
        intr.$O\
        virtio.$O\
        input.$O\
        kbd.$O\
        mouse.$O\
        gpu.$O\
        archvirtriscv.$O\
        main.$O\
        $RISCVOBJ\
        $IP\
        $DEVS\
        $ETHERS\
        $LINKS\
        $PORT\
        $MISC\
        $OTHERS\
        $CONF.root.$O\

LIBNAMES=${LIBS:%=lib%.a}
LIBDIRS=$LIBS

CFLAGS=-wFV -I$ROOT/Inferno/$OBJTYPE/include -I$ROOT/include -I$ROOT/libinterp
KERNDATE=`{$NDATE}

default:V: i$CONF

i$CONF: $OBJ $CONF.c $CONF.root.h $LIBNAMES
        $CC $CFLAGS -DKERNDATE=$KERNDATE $CONF.c
        $LD -l -o $target -H5 -T$LOADADDR $OBJ $CONF.$O $LIBFILES

install:V: i$CONF
        cp i$CONF $INSTALLDIR/i$CONF

<../port/portmkfile

trap.$O: csr.h

main.$O: $ROOT/Inferno/$OBJTYPE/include/ureg.h csr.h
```

```

csr.h csr.s: generate_csr.sh csrregs.h
	sh generate_csr.sh csrregs.h

csr.$O: csr.h csr.s

devuart.$O:	../port/devuart.c ../port/uart.h
	$CC $CFLAGS ../port/devuart.c
```

C /os/virtriscv/main.c

```c
#include "u.h"
#include "../port/lib.h"
#include "dat.h"
#include "mem.h"
#include "fns.h"
#include "../port/uart.h"
#include "sbi.h"
#include "virtio.h"
#include "version.h"

#define MAXCONF         32

Conf conf;
Mach *m = (Mach*)MACHADDR;
Proc *up = 0;

char *confname[MAXCONF];
char *confval[MAXCONF];
int nconf;

extern int main_pool_pcnt;
extern int heap_pool_pcnt;
extern int image_pool_pcnt;

extern freginit(void);

/* Unimplemented functions */
void    fpinit(void) {}
void    FPsave(void*) {}
void    FPrestore(void*) {}
int     segflush(void*, ulong) { return 0; }
void    idlehands(void) { return; }
void    setpanic(void) { return; }

int
pcmspecial(char *idstr, ISAConf *isa)
{
        return -1;
}

void
exit(int panic)
{
        if (panic) {
                iprint("PANIC\n");
        }

        SBI_SHUTDOWN();
        for (;;);
}

void
reboot(void)
{
        spllo();
        print("Rebooting\n");
        (*(volatile unsigned char*)(0x0000)) = 1;
}

void
halt(void)
{
        spllo();
        print("CPU halted\n");
        while (1) {
            wait_for_interrupt();
        }
}

void
addconf(char *name, char *val)
{
        if(nconf >= MAXCONF)
                return;
        confname[nconf] = name;
        confval[nconf] = val;
        nconf++;
}

char*
getconf(char *name)
{
        int i;

        for(i = 0; i < nconf; i++)
                if(cistrcmp(confname[i], name) == 0)
                        return confval[i];
        return 0;
}

void
confinit(void)
{
        ulong base;
        conf.topofmem = 128*MiB + RAMBOOT;

        base = PGROUND((ulong)end);
        conf.base0 = base;

        conf.npage1 = 0;
        conf.npage0 = (conf.topofmem - base)/BY2PG;
        conf.npage = conf.npage0 + conf.npage1;
        conf.ialloc = (((conf.npage*(main_pool_pcnt))/100)/2)*BY2PG;

        conf.nproc = 100 + ((conf.npage*BY2PG)/MB)*5;
        conf.nmach = MAXMACH;

        print("Conf: top=0x%lux, npage0=0x%lux, ialloc=0x%lux, nproc=0x%lux\n",
                        conf.topofmem, conf.npage0,
                        conf.ialloc, conf.nproc);
}

void
poolsizeinit(void)
{
        u64int nb;
        nb = conf.npage*BY2PG;
        poolsize(mainmem, (nb*main_pool_pcnt)/100, 0);
        poolsize(heapmem, (nb*heap_pool_pcnt)/100, 0);
        poolsize(imagmem, (nb*image_pool_pcnt)/100, 1);
}

void
init0(void)
{
        Osenv *o;
        char buf[2*KNAMELEN];

        up->nerrlab = 0;

        print("Starting init0()\n");
        spllo();

        if(waserror())
                panic("init0 %r");

        o = up->env;
        o->pgrp->slash = namec("#/", Atodir, 0, 0);
        cnameclose(o->pgrp->slash->name);
        o->pgrp->slash->name = newcname("/");
        o->pgrp->dot = cclone(o->pgrp->slash);

        chandevinit();

        if(!waserror()){
                ksetenv("cputype", "riscv", 0);
                snprint(buf, sizeof(buf), "riscv %s", conffile);
                ksetenv("terminal", buf, 0);
                poperror();
        }
```

```

	poperror();

	disinit("/osinit.dis");
}

void
userinit(void)
{
	Proc *p;
	Osenv *o;

	p = newproc();
	o = p->env;

	o->fgrp = newfgrp(nil);
	o->pgrp = newpgrp();
	o->egrp = newegrp();
	kstrdup(&o->user, eve);

	strcpy(p->text, "interp");

	p->fpstate = FPINIT;

	p->sched.pc = (ulong)init0;
	p->sched.sp = (ulong)p->kstack+KSTACK-8;

	ready(p);
}

int
main() {
	char input;

	memset(edata, 0, end-edata);
	memset(m, 0, sizeof(Mach));

	freginit();
	confinit();
	xinit();
	poolinit();
	poolsizeinit();

	trapinit();
	clockinit();
	printinit();
	i8250console();
	serwrite = uartputs;
	virtio_init();
	input_init();
	screeninit();

	print("\nRISC-V QEMU\n");
	print("Inferno OS %s Vita Nuova\n\n", VERSION);

	procinit();
	links();
	chandevreset();

	eve = strdup("inferno");

	userinit();
	schedinit();

	halt();
	return 0;
}
```

D /os/virtriscv/sdvirtblk.c

```
#include "u.h"
#include "../port/lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "io.h"
#include "virtio.h"

#include "../port/sd.h"

extern SDifc sdvirtblkifc;

SDev *head;

static int
blk_virtq_init(virtio_dev *dev)
{
	if (dev->queues == 0) {
		dev->queues = malloc(sizeof(virtq));
		dev->numqueues = 1;

		if (dev->queues == 0) {
			panic("Virtio blk: Could not allocate queues. Malloc failed\n");
		}

		if (virtq_alloc(dev, 0, 0) != 0) {
			panic("Virtio blk: Failed to create event queue");
			return -1;
		}

		dev->queues[0].default_handler = nil;
		dev->queues[0].default_handler_data = dev;
	}

	return 0;
}

static int
blk_enable(SDev* sdev)
{
	virtio_enable_interrupt(sdev->ctlr, nil);

	return 1;
}

static int
blk_disable(SDev* sdev)
{
	virtio_disable_interrupt(sdev->ctlr);

	return 1;
}

static SDev*
blk_pnp(void)
{
	virtio_dev *dev;
	SDev *sdev;
	SDev **next;

	for (next = &head; *next != 0; *next = (*next)->next) {}

	while ((dev = virtio_get_device(VIRTIO_DEV_BLOCK)) != 0) {
		int err = virtio_setup(dev, "BLK", blk_virtq_init, VIRTIO_F_ANY_LAYOUT
					| VIRTIO_F_RING_INDIRECT_DESC | VIRTIO_F_RING_EVENT_IDX
					| VIRTIO_BLK_F_RO | VIRTIO_BLK_F_SIZE_MAX | VIRTIO_BLK_F_SEG_MAX);

		switch (err) {
		case 0:
			sdev = malloc(sizeof(SDev));
			sdev->ctlr = dev;
			sdev->ifc = &sdvirtblkifc;
			sdev->nunit = 1;

```

```
				*next = sdev;
				next = &sdev->next;

				blk_enable(sdev);

				break;
			case -1:
				iprint("Virtio blk rejected features\n");
				break;
			case -2:
				iprint("Virtio blk queue error\n");
				break;
			default:
				iprint("Virtio blk unknown error during setup %d\n", err);
				break;
			}
	}

	return head;
}

static SDev*
blk_id(SDev* sdev)
{
	char name[16];
	virtio_dev *dev;
	static char idno[16] = "0123456789";

	for (int i = 0; sdev != nil; sdev = sdev->next) {
		if (sdev->ifc == &sdvirtblkifc) {
			sdev->idno = idno[i++];

			snprint(name, sizeof(name), "virtblk%c", sdev->idno);
			kstrdup(&sdev->name, name);
		}
	}

	return nil;
}

static int
blk_verify(SDunit *unit)
{
	virtio_dev *dev = unit->dev->ctlr;

	snprint((void*) &unit->inquiry[8], sizeof(unit->inquiry)-8,
		"VIRTIO port %d Block Device", dev->index);

	unit->inquiry[4] = sizeof(unit->inquiry)-4;

	return 1;
}

static int
blk_online(SDunit *unit)
{
	virtio_dev *dev = (virtio_dev*) unit->dev->ctlr;
	virtio_blk_config *config = (virtio_blk_config*) &dev->regs->config;

	if (dev->features & VIRTIO_BLK_F_BLK_SIZE) {
		unit->secsize = config->blk_size;
	} else {
		unit->secsize = 512;
	}

	unit->sectors = config->capacity;

	return 1;
}

static long
blk_bio(SDunit* unit, int lun, int write, void* data, long nb, long bno)
{
	ulong len = nb * unit->secsize;
	virtio_dev *dev = (virtio_dev*) unit->dev->ctlr;
	virtio_blk_config *config = (virtio_blk_config*) &dev->regs->config;
```

```
        virtio_blk_req *req;
        uchar *status;

        if (write && dev->features & VIRTIO_BLK_F_RO) {
                // The drive is read-only
                iprint("VIRTIO block write of read only device\n");
                return -1;
        } else if (bno + nb > config->capacity) {
                // Out of bounds
                iprint("VIRTIO block device %s out of bounds\n", write ? "Write" : "Read");
                return -1;
        }

        req = malloc(sizeof(*req));
        req->type = write ? VIRTIO_BLK_T_OUT : VIRTIO_BLK_T_IN;
        req->sector = (bno * unit->secsize) / 512; // VIRTIO always uses sectors of 512, though the device might
        ↪  not
        status = &req->status;
        *status = 255;

        virtq_add_desc_chain(&dev->queues[0], nil, nil, 3,
                             req, VIRTIO_BLK_HDR_SIZE, 0,
                             data, len, write ? 0 : 1,
                             status, VIRTIO_BLK_STATUS_SIZE, 1);

        virtq_make_available(&dev->queues[0]);
        virtq_notify(dev, 0, 0, -1);

        // Block until the drive responds
        while (*status == 255) {}

        switch (*status) {
        case VIRTIO_BLK_S_OK:
                free(req);
                return len;
                break;
        case VIRTIO_BLK_S_IOERR:
                iprint("VIRTIO block device IO error\n");
                break;
        case VIRTIO_BLK_S_UNSUPP:
                iprint("VIRTIO block device unsupported operation\n");
                break;
        default:
                iprint("VIRTIO block device returned %d\n", status);
                error("Unknown VIRTIO block device return code\n");
        }

        free(req);
        return -1;
}

SDifc sdvirtblkifc = {
        "virtblk",                      /* name */

        blk_pnp,                        /* pnp */
        nil,                            /* legacy */
        blk_id,                         /* id */
        blk_enable,                     /* enable */
        blk_disable,                    /* disable */

        blk_verify,                     /* verify */
        blk_online,                     /* online */
        nil,                            /* rio */
        nil,                            /* rctl */
        nil,                            /* wctl */

        blk_bio,                        /* bio */
};
```

E /libinterp/comp-riscv.c

```c
#include "lib9.h"
#include "isa.h"
#include "interp.h"
#include "raise.h"

/*
 * JIT compiler to RISC-V.
 * Assumes that processor supports at least rv32mfd.
 *
 * Note that the operand order is different than the JIT compilers
 * for other architectures, both for instructions and functions.
 * The general order is rd, rs, imm. The exception is store instructions,
 * which goes against the instruction operand ordering by having the source
 * register first.
 */

enum {
        R0      = 0,
        R1      = 1,
        R2      = 2,
        R3      = 3,
        R4      = 4,
        R5      = 5,
        R6      = 6,
        R7      = 7,
        R8      = 8,
        R9      = 9,
        R10     = 10,
        R11     = 11,
        R12     = 12,
        R13     = 13,
        R14     = 14,
        R15     = 15,

        Rlink   = 1,
        Rsp     = 2,
        Rarg    = 8,

        // Temporary registers
        Rtmp    = 4, // Used for building constants and other single-instruction values
        Rta     = 5, // Used for intermediate addresses for double indirect

        // Permanent registers
        Rh      = 6, // Contains H, which is used to check if values are invalid

        // Registers for storing arguments and other mid-term values
        RA0     = 8,
        RA1     = 9,
        RA2     = 10,
        RA3     = 11,
        RA4     = 12,

        Rfp     = 13, // Frame pointer
        Rmp     = 14, // Module pointer
        Rreg    = 15, // Pointer to the REG struct

        // Floating-point registers
        F0      = 0,
        F1      = 1,
        F2      = 2,
        F3      = 3,
        F4      = 4,
        F5      = 5,
        F6      = 6,

        // Opcodes
        OP              = 51,           // 0b0110011
        OPimm           = 19,           // 0b0010011
        OPfp            = 83,           // 0b1010011
        OPlui           = 55,           // 0b0110111
        OPauipc         = 23,           // 0b0010111
        OPjal           = 111,          // 0b1101111
        OPjalr          = 103,          // 0b1100111
        OPbranch        = 99,           // 0b1100011
        OPload          = 3,            // 0b0000011
        OPloadfp        = 7,            // 0b0000111
        OPstore         = 35,           // 0b0100011
        OPstorefp       = 39,           // 0b0100111
        OPmiscmem       = 15,           // 0b0001111
        OPsystem        = 115,          // 0b1110011
        OPamo           = 47,           // 0b0101111
        OPmadd          = 67,           // 0b1000011
        OPnmadd         = 79,           // 0b1001111
        OPmsub          = 71,           // 0b1000111
        OPnmsub         = 75,           // 0b1001011

        // Rounding modes
        RNE             = 0, // Round to nearest, ties to even
        RTZ             = 1, // Round towards zero
        RDN             = 2, // Round down
        RUP             = 3, // Round up
        RMM             = 4, // Round to nearest, ties to max magnitude
        RDYN            = 7, // Use default

        RM              = RDYN, // Default rounding mode

        // Flags to mem
        Ldw = 1,        // Load 32-bit word
        Ldh,            // Load 16-bit half-word (with sign-extension)
        Ldb,            // Load 8-bit byte (with sign-extension)
        Ldhu,           // Load 16-bit unsiged half-word
        Ldbu,           // Load 8-bit unsigned byte
        Lds,            // Load 32-bit single-precision float
        Ldd,            // Load 64-bit double-precision float

        Stw,            // Store 32-bit word
        Sth,            // Store 16-bit half-word
        Stb,            // Store 8-bit byte
        Sts,            // Store 32-bit single-precision float
        Std,            // Store 64-bit double-precision float

        Laddr,          // Special flag for operand functions
                        // Moves the address of the operand to a register

        // Flags to branch
        EQ = 1,
        NE,
        LT,
        LE,
        GT,
        GE,

        // Flags to punt
        SRCOP   = (1<<0),
        DSTOP   = (1<<1),
        WRTPC   = (1<<2),
        TCHECK  = (1<<3),
        NEWPC   = (1<<4),
        DBRAN   = (1<<5),
        THREOP  = (1<<6),

        // The index of each macro
        MacFRP  = 0,
        MacRET,
        MacCASE,
        MacCOLR,
        MacMCAL,
        MacFRAM,
        MacMFRA,
        MacRELQ,
        NMACRO
};

// Masks for the high and low portions of immidiate values
#define IMMSIGNED       0xFFFFF800
#define IMMH            0xFFFFF000
#define IMML            0x00000FFF

// Check if a immidiate has to be split over multiple instructions
#define SPLITIMM(imm)                                   ((((((ulong)(imm)) & IMMSIGNED) != 0)) && ((((ulong)(imm))
↪  & IMMSIGNED) != IMMSIGNED))
```

```c
#define SPLITH(imm)                                     ((((ulong)(imm)) + ((((ulong)(imm)) & (1<<11)) ? (1<<12) :
↪  0)) & IMMH)
#define SPLITL(imm)                                     (((ulong)(imm)) & IMML)

// Extract bits of immidiate values, like imm[11:5]. Basically shifts to the right and masks
// Examples:
// imm[11:5] -> IMM(imm, 11, 5)
// imm[11:0] -> IMM(imm, 11, 0)
// imm[11]   -> IMM(imm, 11, 11)
#define IMM(imm, to, from)                              ((((ulong)(imm)) >> (from)) & ((1 << ((to)-(from)+1)) -
↪  1))

// All RISC-V instruction encoding variants. Set up with LSB on the left and MSB on the right, opposite to the
↪  tables in the RISC-V specification

#define Iimm(imm)                                       (IMM(imm, 11, 0)<<20)
#define Simm(imm)                                       ((IMM(imm, 4, 0)<<7) | (IMM(imm, 11, 5)<<25))
#define Bimm(imm)                                       ((IMM(imm, 11, 11)<<7) | (IMM(imm, 4, 1)<<8) | (IMM(imm,
↪  10, 5)<<25) | (IMM(imm, 12, 12)<<30))
#define Uimm(imm)                                       ((IMM(imm, 31, 12)<<12))
#define Jimm(imm)                                       ((IMM(imm, 19, 12)<<12) | (IMM(imm, 11, 11)<<20) |
↪  (IMM(imm, 10, 1)<<21) | (IMM(imm, 20, 20)<<30))

#define Rtype(op, funct3, funct7, rd, rs1, rs2)         gen((op) | ((rd)<<7) | ((funct3)<<12) | ((rs1)<<15) |
↪  ((rs2)<<20) | ((funct7)<<25))
#define R4type(op, funct3, funct2, rd, rs1, rs2, rs3)   gen((op) | ((rd)<<7) | ((funct3)<<12) | ((rs1)<<15) |
↪  ((rs2)<<20) | ((funct2)<<25) | ((rs3)<<27))
#define Itype(op, funct3, rd, rs1, imm)                 gen((op) | ((rd)<<7) | ((funct3)<<12) | ((rs1)<<15) |
↪  Iimm(imm))
#define Stype(op, funct3, rs1, rs2, imm)                gen((op) | ((funct3)<<12) | ((rs1)<<15) | ((rs2)<<20) |
↪  Simm(imm))
#define Btype(op, funct3, rs1, rs2, imm)                gen((op) | ((funct3)<<12) | ((rs1)<<15) | ((rs2)<<20) |
↪  Bimm(imm))
#define Utype(op, rd, imm)                              gen((op) | ((rd)<<7) | Uimm(imm))
#define Jtype(op, rd, imm)                              gen((op) | ((rd)<<7) | Jimm(imm))

/* Macros for laying down RISC-V instructions. Uses the instruction name from the specification */

// Upper immediate instructions
#define LUI(dest, imm)                                  Utype(OPlui, dest, imm)
#define AUIPC(dest, imm)                                Utype(OPauipc, dest, imm)

// Jump instructions
#define JAL(dest, offset)                               Jtype(OPjal, dest, offset)
#define JALR(dest, base, offset)                        Itype(OPjalr, 0, dest, base, offset)

// Branch instructions
#define BEQ(src1, src2, offset)                         Btype(OPbranch, 0, src1, src2, offset)
#define BNE(src1, src2, offset)                         Btype(OPbranch, 1, src1, src2, offset)
#define BLT(src1, src2, offset)                         Btype(OPbranch, 4, src1, src2, offset)
#define BGE(src1, src2, offset)                         Btype(OPbranch, 5, src1, src2, offset)
#define BLTU(src1, src2, offset)                        Btype(OPbranch, 6, src1, src2, offset)
#define BGEU(src1, src2, offset)                        Btype(OPbranch, 7, src1, src2, offset)

// Load instructions
#define LB(dest, base, imm)                             Itype(OPload, 0, dest, base, imm)
#define LH(dest, base, imm)                             Itype(OPload, 1, dest, base, imm)
#define LW(dest, base, imm)                             Itype(OPload, 2, dest, base, imm)
#define LBU(dest, base, imm)                            Itype(OPload, 4, dest, base, imm)
#define LHU(dest, base, imm)                            Itype(OPload, 5, dest, base, imm)

// Store instructions
#define SB(src, base, imm)                              Stype(OPstore, 0, base, src, imm)
#define SH(src, base, imm)                              Stype(OPstore, 1, base, src, imm)
#define SW(src, base, imm)                              Stype(OPstore, 2, base, src, imm)

// Arithmetic immediate instructions
#define ADDI(dest, src, imm)                            Itype(OPimm, 0, dest, src, imm)
#define SLTI(dest, src, imm)                            Itype(OPimm, 2, dest, src, imm)
#define SLTIU(dest, src, imm)                           Itype(OPimm, 3, dest, src, imm)
#define XORI(dest, src, imm)                            Itype(OPimm, 4, dest, src, imm)
#define ORI(dest, src, imm)                             Itype(OPimm, 6, dest, src, imm)
#define ANDI(dest, src, imm)                            Itype(OPimm, 7, dest, src, imm)

#define SLLI(dest, src, shamt)                          Itype(OPimm, 1, dest, src, shamt)
```

```
#define SRLI(dest, src, shamt)                        Itype(OPimm, 5, dest, src, shamt)
#define SRAI(dest, src, shamt)                        Itype(OPimm, 5, dest, src, shamt | (1<<10))

// Arithmetic register instructions
#define ADD(dest, src1, src2)                         Rtype(OP, 0, 0, dest, src1, src2)
#define SUB(dest, src1, src2)                         Rtype(OP, 0, (1<<5), dest, src1, src2)
#define SLL(dest, src1, src2)                         Rtype(OP, 1, 0, dest, src1, src2)
#define SLT(dest, src1, src2)                         Rtype(OP, 2, 0, dest, src1, src2)
#define SLTU(dest, src1, src2)                        Rtype(OP, 3, 0, dest, src1, src2)
#define XOR(dest, src1, src2)                         Rtype(OP, 4, 0, dest, src1, src2)
#define SRL(dest, src1, src2)                         Rtype(OP, 5, 0, dest, src1, src2)
#define SRA(dest, src1, src2)                         Rtype(OP, 5, (1<<5), dest, src1, src2)
#define OR(dest, src1, src2)                          Rtype(OP, 6, 0, dest, src1, src2)
#define AND(dest, src1, src2)                         Rtype(OP, 7, 0, dest, src1, src2)

// The M extension for multiplication and division
#define MUL(dest, src1, src2)                         Rtype(OP, 0, 1, dest, src1, src2)
#define MULH(dest, src1, src2)                        Rtype(OP, 1, 1, dest, src1, src2)
#define MULHSU(dest, src1, src2)                      Rtype(OP, 2, 1, dest, src1, src2)
#define MULHU(dest, src1, src2)                       Rtype(OP, 3, 1, dest, src1, src2)
#define DIV(dest, src1, src2)                         Rtype(OP, 4, 1, dest, src1, src2)
#define DIVU(dest, src1, src2)                        Rtype(OP, 5, 1, dest, src1, src2)
#define REM(dest, src1, src2)                         Rtype(OP, 6, 1, dest, src1, src2)
#define REMU(dest, src1, src2)                        Rtype(OP, 7, 1, dest, src1, src2)

// The F extension for single-precision floating-point. rm is the rounding mode
#define FLW(dest, base, offset)                       Itype(OPloadfp, 2, dest, base, offset)
#define FSW(src, base, offset)                        Stype(OPstorefp, 2, base, src, offset)

#define FMADDS(rm, dest, src1, src2, src3)            R4type(OPmadd, rm, 0, dest, src1, src2, src3)
#define FMSUBS(rm, dest, src1, src2, src3)            R4type(OPmsub, rm, 0, dest, src1, src2, src3)
#define FNMADDS(rm, dest, src1, src2, src3)           R4type(OPnmadd, rm, 0, dest, src1, src2, src3)
#define FNMSUBS(rm, dest, src1, src2, src3)           R4type(OPnmsub, rm, 0, dest, src1, src2, src3)

#define FADDS(rm, dest, src1, src2)                   Rtype(OPfp, rm, 0, dest, src1, src2)
#define FSUBS(rm, dest, src1, src2)                   Rtype(OPfp, rm, 1<<2, dest, src1, src2)
#define FMULS(rm, dest, src1, src2)                   Rtype(OPfp, rm, 1<<3, dest, src1, src2)
#define FDIVS(rm, dest, src1, src2)                   Rtype(OPfp, rm, 3<<2, dest, src1, src2)
#define FSQRTS(rm, dest, src)                         Rtype(OPfp, rm, 11<<2, dest, src, 0)

#define FSGNJS(dest, src1, src2)                      Rtype(OPfp,  0, 1<<4, dest, src1, src2)
#define FSGNJNS(dest, src1, src2)                     Rtype(OPfp,  1, 1<<4, dest, src1, src2)
#define FSGNJXS(dest, src1, src2)                     Rtype(OPfp,  2, 1<<4, dest, src1, src2)

#define FMINS(dest, src1, src2)                       Rtype(OPfp,  0, 5<<2, dest, src1, src2)
#define FMAXS(dest, src1, src2)                       Rtype(OPfp,  1, 5<<2, dest, src1, src2)

#define FMVXW(dest, src)                              Rtype(OPfp,  0, 7<<4, dest, src, 0)
#define FMVWX(rm, dest, src)                          Rtype(OPfp,  0, 15<<3, dest, src, 0)

#define FEQS(dest, src1, src2)                        Rtype(OPfp,  2, 5<<4, dest, src1, src2)
#define FLTS(dest, src1, src2)                        Rtype(OPfp,  1, 5<<4, dest, src1, src2)
#define FLES(dest, src1, src2)                        Rtype(OPfp,  0, 5<<4, dest, src1, src2)

#define FCLASSS(dest, src)                            Rtype(OPfp,  1, 7<<4, dest, src, 0)

#define FCVTWS(rm, dest, src)                         Rtype(OPfp, rm, 3<<5, dest, src, 0)
#define FCVTWUS(rm, dest, src)                        Rtype(OPfp, rm, 3<<5, dest, src, 1)
#define FCVTSW(rm, dest, src)                         Rtype(OPfp, rm, 13<<3, dest, src, 0)
#define FCVTSWU(rm, dest, src)                        Rtype(OPfp, rm, 13<<3, dest, src, 1)

// The D extension for double-precision floating-point
#define FLD(dest, base, offset)                       Itype(OPloadfp, 3, dest, base, offset)
#define FSD(src, base, offset)                        Stype(OPstorefp, 3, base, src, offset)

#define FMADDD(rm, dest, src1, src2, src3)            R4type(OPmadd, rm, 1, dest, src1, src2, src3)
#define FMSUBD(rm, dest, src1, src2, src3)            R4type(OPmsub, rm, 1, dest, src1, src2, src3)
#define FNMADDD(rm, dest, src1, src2, src3)           R4type(OPnmadd, rm, 1, dest, src1, src2, src3)
#define FNMSUBD(rm, dest, src1, src2, src3)           R4type(OPnmsub, rm, 1, dest, src1, src2, src3)

#define FADDD(rm, dest, src1, src2)                   Rtype(OPfp, rm, 1, dest, src1, src2)
#define FSUBD(rm, dest, src1, src2)                   Rtype(OPfp, rm, 5, dest, src1, src2)
#define FMULD(rm, dest, src1, src2)                   Rtype(OPfp, rm, 9, dest, src1, src2)
#define FDIVD(rm, dest, src1, src2)                   Rtype(OPfp, rm, 13, dest, src1, src2)
#define FSQRTD(rm, dest, src)                         Rtype(OPfp, rm, 45, dest, src, 0)
```

```
#define FSGNJD(dest, src1, src2)                         Rtype(OPfp,  0, 17, dest, src1, src2)
#define FSGNJND(dest, src1, src2)                        Rtype(OPfp,  1, 17, dest, src1, src2)
#define FSGNJXD(dest, src1, src2)                        Rtype(OPfp,  2, 17, dest, src1, src2)

#define FMIND(dest, src1, src2)                          Rtype(OPfp,  0, 21, dest, src1, src2)
#define FMAXD(dest, src1, src2)                          Rtype(OPfp,  1, 21, dest, src1, src2)

#define FEQD(dest, src1, src2)                           Rtype(OPfp,  2, 81, dest, src1, src2)
#define FLTD(dest, src1, src2)                           Rtype(OPfp,  1, 81, dest, src1, src2)
#define FLED(dest, src1, src2)                           Rtype(OPfp,  0, 81, dest, src1, src2)

#define FCLASSD(dest, src)                               Rtype(OPfp,  1, 113, dest, src, 0)

#define FCVTSD(rm, dest, src)                            Rtype(OPfp, rm, 32, dest, src, 1)
#define FCVTDS(rm, dest, src)                            Rtype(OPfp, rm, 32, dest, src, 0)
#define FCVTWD(rm, dest, src)                            Rtype(OPfp, rm, 97, dest, src, 0)
#define FCVTWUD(rm, dest, src)                           Rtype(OPfp, rm, 97, dest, src, 1)
#define FCVTDW(rm, dest, src)                            Rtype(OPfp, rm, 105, dest, src, 0)
#define FCVTDWU(rm, dest, src)                           Rtype(OPfp, rm, 105, dest, src, 1)

// Pseudoinstructions
#define MOV(rd, rs)                                      ADDI(rd, rs, 0)
#define NOT(rd, rs)                                      XORI(rd, rs, -1)
#define NEG(rd, rs)                                      SUB(rd, R0, rs)

#define BEQZ(rs, offset)                                 BEQ(rs, R0, offset)
#define BNEZ(rs, offset)                                 BNE(rs, R0, offset)
#define BLEZ(rs, offset)                                 BGE(R0, rs, offset)
#define BGEZ(rs, offset)                                 BGE(rs, R0, offset)
#define BLTZ(rs, offset)                                 BLT(rs, R0, offset)
#define BGTZ(rs, offset)                                 BLT(R0, rs, offset)

#define BGT(rs1, rs2, offset)                            BLT(rs2, rs1, offset)
#define BLE(rs1, rs2, offset)                            BGE(rs2, rs1, offset)
#define BGTU(rs1, rs2, offset)                           BLTU(rs2, rs1, offset)
#define BLEU(rs1, rs2, offset)                           BGEU(rs2, rs1, offset)

#define JUMP(offset)                                     JAL(R0, offset)
#define JL(offset)                                       JAL(R1, offset)
#define JR(rs, offset)                                   JALR(R0, rs, offset)
#define JRL(rs, offset)                                  JALR(R1, rs, offset)

/* Helper macros */

// Used to look up the address of an array element relative to base
#define IA(s, o)                        (ulong)(base+s[o])

// The offset from the current code address to the pointer
#define OFF(ptr)                        ((ulong)(ptr) - (ulong) (code))

// Call a function at the given address
#define CALL(o)                         (LUI(Rtmp, SPLITH(o)), JRL(Rtmp, SPLITL(o)))

// Return from a function
#define RETURN                          JR(Rlink, 0)

// Call a macro. Takes the macro idx as the argument
#define CALLMAC(idx)                    CALL(IA(macro, idx))

// Jump to a specific address
#define JABS(ptr)                       (LUI(Rtmp, SPLITH(ptr)), JR(Rtmp, SPLITL(ptr)))

// Jump to a Dis address
#define JDIS(pc)                        JABS(IA(patch, pc))

// Jump to an address in the dst field of an instruction
#define JDST(i)                         JDIS((i->d.ins - mod->prog))

// Set the offset of a branch instruction at address ptr to the current code address
// The order is opposite from OFF because it is used where the branch should jump to,
// not where it jumps from
#define PATCHBRANCH(ptr)                *ptr |= Bimm((ulong)(code) - (ulong)(ptr))

// Gets the address of a PC relative to the base
#define RELPC(pc)                       (ulong)(base+(pc))
```

```
// Throw an error if the register is 0
#define NOTNIL(r)                       (BNE(r, R0, 12), LUI(Rtmp, nullity), JRL(Rtmp, nullity))

// Array bounds check. Throws an error if the index is out of bounds
#define BCK(rindex, rsize)              (BLTU(rindex, rsize, 8), /*CALL(bounds)*/ CRASH())

// Cause an immediate illegal instruction exception, which should
// cause a register dump, stack trace, and a halt.
// Useful for debugging register state
#define CRASH()                         gen(0)


static  ulong*  code;
static  ulong*  codestart;
static  ulong*  codeend;
static  ulong*  base;
static  ulong*  patch;
static  ulong   codeoff;
static  int     pass;
static  int     puntpc = 1;
static  Module* mod;
static  uchar*  tinit;
static  ulong*  litpool;
static  int     nlit;
static  ulong   macro[NMACRO];
        void    (*comvec)(void);
static  void    macfrp(void);
static  void    macret(void);
static  void    maccase(void);
static  void    maccolr(void);
static  void    macmcal(void);
static  void    macfram(void);
static  void    macmfra(void);
static  void    macrelq(void);
static  void movmem(Inst*);
static  void mid(Inst*, int, int);

extern  void    das(ulong*, int);
extern  void    _d2v(vlong *y, double d);

// Float constants
double double05 = 0.5;
double double4294967296 = 4294967296.0;

#define T(r)    *((void**)(R.r))

// The macro table. Macros are long sequences of instructions which come up often, like calls and returns,
// so they are extracted out into separate blocks. The calling convention is separate for each macro.
struct
{
        int     idx;
        void    (*gen)(void);
        char*   name;
} mactab[] =
{
        MacFRP,         macfrp,         "FRP",  /* decrement and free pointer */
        MacRET,         macret,         "RET",  /* return instruction */
        MacCASE,        maccase,        "CASE", /* case instruction */
        MacCOLR,        maccolr,        "COLR", /* increment and color pointer */
        MacMCAL,        macmcal,        "MCAL", /* mcall bottom half */
        MacFRAM,        macfram,        "FRAM", /* frame instruction */
        MacMFRA,        macmfra,        "MFRA", /* punt mframe because t->initialize==0 */
        MacRELQ,        macrelq,        "RELQ", /* reschedule */
};


/*  Helper functions */

void
urk(char *s)
{
        iprint("urk: %s\n", s);
        error(exCompile);
}

static void
```

```
gen(u32int o)
{
	if (code < codestart || code >= codeend) {
		iprint("gen: code out of bounds\n");
		iprint("code:      0x%p\n", code);
		iprint("codestart: 0x%p\n", codestart);
		iprint("codeend:   0x%p\n", codeend);
		//while (1) {}
	}

	*code++ = o;
}

static void
loadi(int reg, ulong val)
{
	// Load a value into a register

	// Check if the upper 20 bits are needed
	if (SPLITIMM(val)) {
		// Check if the lower 12 bits are needed
		LUI(reg, SPLITH(val));
		ADDI(reg, reg, SPLITL(val));
	} else {
		ADDI(reg, R0, val);
	}
}

static void
multiply(int rd, int rs, long c)
{
	// Multiply by a constant, rd = rs * c
	int shamt;

	if (c < 0) {
		NEG(rd, rs);
		rs = rd;
		c = -c;
	}

	switch (c) {
	case 0:
		MOV(rd, R0);
		break;
	case 1:
		if (rd != rs)
			MOV(rd, rs);
		break;
	case 2:
		shamt = 1;
		goto shift;
	case 3:
		shamt = 1;
		goto shiftadd;
	case 4:
		shamt = 2;
		goto shift;
	case 5:
		shamt = 2;
		goto shiftadd;
	case 7:
		shamt = 3;
		goto shiftsub;
	case 8:
		shamt = 3;
		goto shift;
	case 16:
		shamt = 4;
		goto shift;
	case 32:
		shamt = 5;
		goto shift;
	case 64:
		shamt = 6;
		goto shift;
	case 128:
			shamt = 7;
			goto shift;
		case 256:
			shamt = 8;
			goto shift;
		case 512:
			shamt = 9;
			goto shift;
		case 1024:
			shamt = 10;
			goto shift;
		shift:
			SLLI(rd, rs, shamt);
			break;
		shiftadd:
			if (rd == rs) {
				MOV(Rtmp, rs);
				rs = Rtmp;
			}

			SLLI(rd, rs, shamt);
			ADD(rd, rd, rs);
			break;
		shiftsub:
			if (rd == rs) {
				MOV(Rtmp, rs);
				rs = Rtmp;
			}

			SLLI(rd, rs, shamt);
			SUB(rd, rd, rs);
			break;
		default:
			loadi(Rtmp, c);
			MUL(rd, rd, Rtmp);
		}
}

static void
mem(int type, int r, int base, long offset)
{
		// Load or store data at an offset from an address in a register.
		// - type should be one of Ld* or St*.
		// - r is the source or destination register.
		// - base is the register with the base address.
		// - offset is added to the value of rs to get the
		//   address to load/store from/to

		if (SPLITIMM(offset)) {
			// The offset is too long. Add the upper part of offset to rs in the tmp register,
			// and use that as the base instead.
			LUI(Rtmp, SPLITH(offset));
			ADD(Rtmp, Rtmp, base);
			base = Rtmp;
			offset = SPLITL(offset);
		}

		switch (type) {
		case Ldw:
			LW(r, base, offset);
			break;
		case Ldh:
			LH(r, base, offset);
			break;
		case Ldhu:
			LHU(r, base, offset);
			break;
		case Ldb:
			LB(r, base, offset);
			break;
		case Ldbu:
			LBU(r, base, offset);
			break;
		case Lds:
			FLW(r, base, offset);
			break;
```

```
	case Ldd:
		FLD(r, base, offset);
		break;
	case Stw:
		SW(r, base, offset);
		break;
	case Sth:
		SH(r, base, offset);
		break;
	case Stb:
		SB(r, base, offset);
		break;
	case Sts:
		FSW(r, base, offset);
		break;
	case Std:
		FSD(r, base, offset);
		break;
	case Laddr:
		ADDI(r, base, offset);
		break;
	default:
		if (cflag > 2)
			iprint("Invalid type argument to mem: %d\n", type);
		urk("mem");
		break;
	}
}

static void
operand(int mtype, int mode, Adr *a, int r, int li)
{
	// Load or store the value from a src or dst operand of an instruction
	// - mtype is the memory access type, as in mem
	// - mode is the mode bits of the operand fields
	// - a is the source or dest struct
	// - r is the register to load the address into
	int base;
	long offset;

	switch (mode) {
	default:
		urk("operand");
	case AIMM:
		// Immediate value
		loadi(r, a->imm);

		if (mtype == Laddr) {
			mem(Stw, r, Rreg, li);
			mem(Laddr, r, Rreg, li);
		}
		return;
	case AFP:
		// Indirect offset from FP
		base = Rfp;
		offset = a->ind;
		break;
	case AMP:
		// Indirect offset from MP
		base = Rmp;
		offset = a->ind;
		break;
	case AIND|AFP:
		// Double indirect from FP
		mem(Ldw, Rta, Rfp, a->i.f);
		base = Rta;
		offset = a->i.s;
		break;
	case AIND|AMP:
		// Double indirect from MP
		mem(Ldw, Rta, Rmp, a->i.f);
		base = Rta;
		offset = a->i.s;
		break;
	}

	mem(mtype, r, base, offset);
}

static void
op1(int mtype, Inst *i, int r)
{
	// Load or store the source operand
	operand(mtype, USRC(i->add), &i->s, r, O(REG, st));
}

static void
op3(int mtype, Inst *i, int r)
{
	// Load or store the dest operand
	operand(mtype, UDST(i->add), &i->d, r, O(REG, dt));
}

static void
op2(int mtype, Inst *i, int r)
{
	// Load or store the middle operand
	int ir;

	switch (i->add&ARM) {
	default:
		return;
	case AXIMM:
		// Short immediate
		loadi(r, (short) i->reg);

		if (mtype == Laddr) {
			mem(Stw, r, Rreg, O(REG, t));
			mem(Laddr, r, Rreg, O(REG, t));
		}
		return;
	case AXINF:
		// Small offset from FP
		ir = Rfp;
		break;
	case AXINM:
		// Small offset from MP
		ir = Rmp;
		break;
	}

	// Load indirect
	mem(mtype, r, ir, i->reg);
}

static void
literal(ulong imm, int roff)
{
	// TODO: Why do this?
	nlit++;

	loadi(Rta, (ulong) litpool);
	mem(Stw, Rta, Rreg, roff);

	if (pass == 0)
		return;

	*litpool = imm;
	litpool++;
}

static void
rdestroy(void)
{
	destroy(R.s);
}

static void
rmcall(void)
{
	// Called by the compiled code to transfer control during an mcall
	Frame *f;
```

```
        Prog *p;

        if (R.dt == (ulong) H)
                error(exModule);

        f = (Frame*)R.FP;
        if (f == H)
                error(exModule);

        f->mr = nil;

        ((void(*)(Frame*))R.dt)(f);

        R.SP = (uchar*)f;
        R.FP = f->fp;

        if (f->t == nil)
                unextend(f);
        else
                freeptrs(f, f->t);

        p = currun();
        if (p->kill != nil)
                error(p->kill);
}

static void
rmfram(void)
{
        Type *t;
        Frame *f;
        uchar *nsp;

        if(R.d == H)
                error(exModule);
        t = (Type*)R.s;
        if(t == H)
                error(exModule);
        nsp = R.SP + t->size;
        if(nsp >= R.TS) {
                R.s = t;
                extend();
                T(d) = R.s;
                return;
        }
        f = (Frame*)R.SP;
        R.SP = nsp;
        f->t = t;
        f->mr = nil;
        initmem(t, f);
        T(d) = f;
}

static void
bounds(void)
{
        error(exBounds);
}

static void
nullity(void)
{
        error(exNilref);
}

static void
punt(Inst *i, int m, void (*fn)(void))
{
        ulong pc;
        ulong *branch;

        if (m & SRCOP) {
                // Save the src operand in R->s
                op1(Laddr, i, RA1);
                mem(Stw, RA1, Rreg, O(REG, s));
        }
```

```c

	if (m & DSTOP) {
		// Save the dst operand in R->d
		op3(Laddr, i, RA3);
		mem(Stw, RA3, Rreg, O(REG, d));
	}

	if (m & WRTPC) {
		// Store the PC in R->PC
		loadi(RA0, RELPC(patch[i - mod->prog+1]));
		mem(Stw, RA0, Rreg, O(REG, PC));
	}

	if (m & DBRAN) {
		// TODO: What does this do?
		pc = patch[i->d.ins - mod->prog];
		literal((ulong) (base+pc), O(REG, d));
	}

	if ((i->add & ARM) == AXNON) {
		if (m & THREOP) {
			// R->m = R->d
			mem(Ldw, RA2, Rreg, O(REG, d));
			mem(Stw, RA2, Rreg, O(REG, m));
		}
	} else {
		// R->m = middle operand
		op2(Laddr, i, RA2);
		mem(Stw, RA2, Rreg, O(REG, m));
	}

	// R->FP = Rfp
	mem(Stw, Rfp, Rreg, O(REG, FP));

	CALL(fn);

	loadi(Rreg, (ulong) &R);

	if (m & TCHECK) {
		mem(Ldw, RA0, Rreg, O(REG, t));

		branch = code;
		BEQZ(RA0, 0);

		// If R->t != 0
		mem(Ldw, Rlink, Rreg, O(REG, xpc)); // Rlink = R->xpc
		RETURN;

		PATCHBRANCH(branch); // endif
	}

	mem(Ldw, Rfp, Rreg, O(REG, FP));
	mem(Ldw, Rmp, Rreg, O(REG, MP));

	if (m & NEWPC) {
		// Jump to R->PC
		mem(Ldw, RA0, Rreg, O(REG, PC));
		JR(RA0, 0);
	}
}

static void
movloop(uint s)
{
	// Move a section of memory in a loop.
	// s is the size of each value, and should be 1, 2, or 4.
	// The source address should be in RA1.
	// The destination address should be in RA2.
	// The amount of values to transfer should be in RA3
	// All registers will be altered

	ulong *loop;

	if (s > 4 && s == 3) {
		// Unnatural size. Transfer byte for byte
		s = 1;
	}

	loop = code;
	BEQZ(RA3, 0);

	switch (s) {
	case 0:
		MOV(RA3, R0);
		break;
	case 1:
		mem(Ldb, RA0, RA1, 0);
		mem(Stb, RA0, RA2, 0);
		break;
	case 2:
		mem(Ldh, RA0, RA1, 0);
		mem(Ldh, RA0, RA2, 0);
		break;
	case 4:
		mem(Ldw, RA0, RA1, 0);
		mem(Ldw, RA0, RA2, 0);
		break;
	default:
		urk("movloop");
	}

	ADDI(RA1, RA2, s);
	ADDI(RA1, RA2, s);
	ADDI(RA3, RA3, -s);

	JABS(loop);

	PATCHBRANCH(loop);
}

static void
movmem(Inst *i)
{
	// Move a region of memory. Makes small transfers efficient, while defaulting
	// to a move loop for larger transfers.
	// The source address should be in RA1
	ulong *branch;

	if ((i->add & ARM) != AXIMM) {
		op2(Ldw, i, RA3);

		branch = code;
		BEQ(RA3, R0, 0);

		// if src2 != 0
		movloop(1);
		// endif

		PATCHBRANCH(branch);
		return;
	}

	switch (i->reg) {
	case 0:
		break;
	case 4:
		mem(Ldw, RA2, RA1, 0);
		op3(Stw, i, RA2); // Save directly, don't bother loading the address
		break;
	case 8:
		mem(Ldw, RA2, RA1, 0);
		mem(Ldw, RA3, RA1, 4);

		op3(Laddr, i, RA4);
		mem(Stw, RA2, RA4, 0);
		mem(Stw, RA3, RA4, 4);
		break;
	default:
		op3(Laddr, i, RA2);

		if ((i->reg & 3) == 0) {
			loadi(RA3, i->reg >> 2);
```

```
				movloop(4);
			} else if ((i->reg & 1) == 0) {
				loadi(RA3, i->reg >> 1);
				movloop(2);
			} else {
				loadi(RA3, i->reg);
				movloop(1);
			}
			break;
	}
}

static void
movptr(Inst *i)
{
	// Arguments:
	// - RA1: The address to move from
	// - op3: The address to move to

	ulong *branch;

	branch = code;
	BEQ(RA1, Rh, 0);

	// if RA1 != H
	CALLMAC(MacCOLR);		// colour if not H
	// endif

	PATCHBRANCH(branch);

	op3(Laddr, i, RA2);
	NOTNIL(RA2);

	mem(Ldw, RA0, RA2, 0);
	mem(Stw, RA1, RA2, 0);
	CALLMAC(MacFRP);
}

static void
branch(Inst *i, int mtype, int btype)
{
	// Insert a branch comparing integers
	// mtype should be the mtype to pass to mem to get the correct width
	// btype should be a constant like EQ, NE, LT, etc
	ulong *branch;

	op2(mtype, i, RA1);
	op1(mtype, i, RA2);

	branch = code;

	// Invert the condition to skip the jump
	switch (btype) {
	case EQ:
		BNE(RA1, RA2, 0);
		break;
	case NE:
		BEQ(RA1, RA2, 0);
		break;
	case GT:
		BLE(RA1, RA2, 0);
		break;
	case LT:
		BGE(RA1, RA2, 0);
		break;
	case LE:
		BGT(RA1, RA2, 0);
		break;
	case GE:
		BLT(RA1, RA2, 0);
		break;
	}

	iprint("branch: pc %d, branch to %d\n", ((ulong)i-(ulong)mod->prog), ((ulong)i->d.ins -
	↪  (ulong)mod->prog));

```

```
        JDST(i);

        PATCHBRANCH(branch);
}

static void
branchl(Inst *i, int btype)
{
        // Insert a branch comparing 64-bit integers
        // btype should be a constant like EQ, NE, LT, etc
        ulong *branch;

        op1(Laddr, i, RA0);
        mem(Ldw, RA1, RA0, 0);
        mem(Ldw, RA2, RA0, 4);

        op2(Laddr, i, RA0);
        mem(Ldw, RA3, RA0, 0);
        mem(Ldw, RA4, RA0, 4);

        // Set RA1 and RA2 to 1 if the condition holds
        switch (btype) {
        case EQ:
        case NE:
                // RA1 = RA1 - RA3 == 0
                // RA2 = RA2 - RA4 == 0
                SUB(RA1, RA1, RA3);
                SUB(RA2, RA2, RA4);
                SLTU(RA1, R0, RA1);
                SLTU(RA2, R0, RA2);
                break;
        case LT:
        case GE:
                // RA1 = RA1 < RA3
                // RA2 = RA2 < RA4
                SLT(RA1, RA1, RA3);
                SLT(RA2, RA2, RA4);
                break;
        case GT:
        case LE:
                // RA1 = RA3 < RA1
                // RA2 = RA4 < RA2
                SLT(RA1, RA3, RA1);
                SLT(RA2, RA4, RA2);
                break;
        }

        AND(RA1, RA1, RA2);

        // Insert the branch. Negate to skip the jump
        // Have to negate again for NE, GE and LE
        branch = code;
        switch (btype) {
        case NE:
        case GE:
        case LE:
                // If the negated condition holds, skip the jump
                BNE(RA1, R0, 0);
                break;
        default:
                // If the condition doesn't hold, skip the jump
                BEQ(RA1, R0, 0);
                break;
        }

        JDST(i);

        PATCHBRANCH(branch);
}

static void
branchfd(Inst *i, int btype)
{
        // Insert a branch comparing double-precision floats
        // btype should be a constant like EQ, NE, LT, etc
        ulong *branch;

        op2(Ldd, i, F1);
        op1(Ldd, i, F2);

        // Float compare instructions don't branch, so the branch
        // instruction has to check the result
        switch (btype) {
        case EQ:
        case NE:
                FEQD(RA0, F1, F2);
                break;
        case LT:
        case GE:
                FLTD(RA0, F1, F2);
                break;
        case LE:
        case GT:
                FLED(RA0, F1, F2);
                break;
        }

        // Branch if the result is negative, skipping the jump
        branch = code;
        switch (btype) {
        case NE:
        case GE:
        case GT:
                BNE(RA0, R0, 0);
                break;
        default:
                BEQ(RA0, R0, 0);
                break;
        }

        JDST(i);

        PATCHBRANCH(branch);
}

/* Macros */
static void
macfram(void)
{
        // Allocate a mframe
        // Arguments:
        // - RA3: src1->links[src2]->t

        ulong *branch;

        mem(Ldw, RA2, Rreg, O(REG, SP));        // RA2 = f = R.SP
        mem(Ldw, RA1, RA3, O(Type, size));      // RA1 = src1->links[src2]->t->size
        ADD(RA0, RA2, RA1);                     // RA0 = nsp = R.SP + t->size
        mem(Ldw, RA1, Rreg, O(REG, TS));        // RA1 = R->TS

        branch = code;
        BGEU(RA0, RA1, 0);

        // nsp < R.TS
        mem(Stw, RA2, Rreg, O(REG, SP));        // R.SP = nsp

        mem(Stw, RA3, RA2, O(Frame, t));        // f->t = RA3
        mem(Stw, R0, RA2, O(Frame, mr));        // f->mr = 0
        mem(Ldw, Rta, RA3, O(Type, initialize));
        JRL(Rta, 0);                            // call t->init(RA2)

        // nsp >= R.TS; must expand
        PATCHBRANCH(branch);
        // Call extend. Store registers
        mem(Stw, RA3, Rreg, O(REG, s));
        mem(Stw, Rlink, Rreg, O(REG, st));
        mem(Stw, Rfp, Rreg, O(REG, FP));
        CALL(extend);

        // Restore registers
        loadi(Rreg, (ulong) &R);
        mem(Ldw, Rlink, Rreg, O(REG, st));
        mem(Ldw, Rfp, Rreg, O(REG, FP));
        mem(Ldw, Rmp, Rreg, O(REG, MP));
        mem(Ldw, RA2, Rreg, O(REG, s));
        RETURN;
}

static void
macmfra(void)
{
        mem(Stw, Rlink, Rreg, O(REG, st));
        mem(Stw, RA3, Rreg, O(REG, s)); // Save type
        mem(Stw, RA0, Rreg, O(REG, d)); // Save destination
        mem(Stw, Rfp, Rreg, O(REG, FP));

        CALL(rmfram);

        loadi(Rreg, (ulong)&R);
        mem(Ldw, Rlink, Rreg, O(REG, st));
        mem(Ldw, Rfp, Rreg, O(REG, FP));
        mem(Ldw, Rmp, Rreg, O(REG, MP));

        RETURN;
}

static void
macmcal(void)
{
        // The bottom half of a mcall instruction
        // Calling convention:
        // - RA0: The address of the function to jump to
        // - RA2: The frame address, src1 to mcall
        // - RA3: The module reference, src3 to mcall

        ulong *branch1, *branch2, *branch3;

        branch1 = code;
        BEQ(RA0, Rh, 0);
        // If RA0 != H

        mem(Ldw, RA1, RA3, O(Modlink, prog));   // Load m->prog into RA1

        branch2 = code;
        BNEZ(RA1, 0);
        // If m->prog == 0

        mem(Stw, Rlink, Rreg, O(REG, st));      // Store link register
        mem(Stw, RA2, Rreg, O(REG, FP));        // Store FP register
        mem(Stw, RA0, Rreg, O(REG, dt));        // Store destination address

        CALL(rmcall);

        // After the call has returned
        loadi(Rreg, (ulong)&R);                 // Load R
        mem(Ldw, Rlink, Rreg, O(REG, st));      // Load link register
        mem(Ldw, Rfp, Rreg, O(REG, FP));        // Load FP register
        mem(Ldw, Rmp, Rreg, O(REG, MP));        // Load MP register
        RETURN;

        // else
        PATCHBRANCH(branch1);                   // If RA0 == H
        PATCHBRANCH(branch2);                   // If m->prog != 0

        MOV(Rfp, RA2);                          // Rfp = RA2
        mem(Stw, RA3, Rreg, O(REG, M));         // R.M = RA3

        // D2H(RA3)->ref++
        ulong heapref = O(Heap, ref) - sizeof(Heap);
        mem(Ldw, RA1, RA3, heapref);
        ADDI(RA1, RA1, 1);
        mem(Stw, RA1, RA3, heapref);

        mem(Ldw, Rmp, RA3, O(Modlink, MP));     // Rmp = R.M->mp
        mem(Stw, Rmp, Rreg, O(REG, MP));        // R.MP = Rmp

        mem(Ldw, RA1, RA3, O(Modlink, compiled));
        branch3 = code;
        BNEZ(RA1, 0);

        // if M.compiled == 0
        mem(Stw, Rfp, Rreg, O(REG, FP)); // R.FP = Rfp
        mem(Stw, RA0, Rreg, O(REG, PC)); // R.PC = Rpc
        mem(Ldw, Rlink, Rreg, O(REG, xpc));
        RETURN;                         // Leave it to the interpreter to handle

        // else
        PATCHBRANCH(branch3);
        JR(RA0, 0);                     // Jump to the compiled module
}

static void
maccase(void)
{
        /*
        * RA1 = value (input arg), v
        * RA2 = count, n
        * RA3 = table pointer (input arg), t
        * RA0 = n/2, n2
        * RA4 = pivot element t+n/2*3, l
        */

        ulong *loop, *found, *branch;

        mem(Ldw, RA2, RA3, 0);          // get count from table
        MOV(Rlink, RA3);                // initial table pointer

        loop = code;
        BLEZ(RA2, 0);                   // n <= 0? goto out

        SRAI(RA0, RA2, 1);              // n2 = n>>1

        // l = t + n/2*3
        ADD(RA4, RA0, RA2);             // l = n/2 + n
        ADD(RA4, RA3, RA1);             // l += t

        mem(Ldw, Rta, RA4, 4);          // Rta = l[1]
        branch = code;
        BGE(RA1, Rta, 0);

        // if v < l[1]
        MOV(RA2, RA0);                  // n = n2
        JABS(loop);                     // continue

        // if v >= l[1]
        PATCHBRANCH(branch);
        mem(Ldw, Rta, RA4, 8);          // Rta = l[2]
        found = code;
        BLT(RA1, Rta, 0);               // branch to found

        // if v >= l[2]
        ADDI(RA3, RA4, 12);             // t = l+3
        SUB(RA2, RA2, RA0);             // n -= n2
        ADDI(RA2, RA2, -1);             // n -= 1

        JABS(loop);                     // goto loop

        // endloop

        // found: v >= l[1] && v < l[2]
        // jump to l[3]
        PATCHBRANCH(found);
        JR(RA4, 12);

        // out: Loop ended
        PATCHBRANCH(loop);
        mem(Ldw, RA2, Rlink, 0);        // load initial n
        ADD(Rtmp, RA2, RA2);            // Rtmp = 2*n
        ADD(RA2, RA2, Rtmp);            // n = 3*n

        // goto (initial t)[n*3+1]
        SLLI(RA2, RA2, 2);              // RA2 = n*sizeof(long)
        ADD(Rlink, Rlink, RA2);         // Rlink = t[n*3]
        JR(Rlink, 4);                   // goto Rlink+4 = t[n*3+1]
```

```
}

static void
maccolr(void)
{
	// Color a pointer
	// Arguments:
	// - RA1: The pointer to color
	ulong *branch;

	// h->ref++
	mem(Ldw, RA0, RA1, O(Heap, ref) - sizeof(Heap));
	ADDI(RA0, RA0, 1);
	mem(Stw, RA0, RA1, O(Heap, ref) - sizeof(Heap));

	// RA0 = mutator
	mem(Ldw, RA0, RA1, O(Heap, color) - sizeof(Heap));

	// RA2 = h->color
	loadi(RA2, (ulong) &mutator);
	mem(Ldw, RA2, RA2, 0);

	branch = code;
	BEQ(RA0, RA2, 0);

	// if h->color != mutator

	// h->color = propagator
	loadi(RA2, propagator);
	mem(Stw, RA2, RA1, O(Heap, color) - sizeof(Heap));

	// nprop = RA1
	loadi(RA2, (ulong) &nprop);
	mem(Stw, RA1, RA2, 0);

	// endif
	PATCHBRANCH(branch);
	RETURN;
}

static void
macfrp(void)
{
	// Destroy a pointer
	// Arguments:
	// - RA0: The pointer to destroy
	ulong *branch1, *branch2;

	branch1 = code;
	BEQ(RA0, Rh, 0);

	// if RA0 != H
	mem(Ldw, RA2, RA0, O(Heap, ref) - sizeof(Heap));
	ADDI(RA2, RA2, -1);

	branch2 = code;
	BEQ(RA2, R0, 0);

	// if --h->ref != 0
	mem(Stw, RA2, RA0, O(Heap, ref) - sizeof(Heap));
	RETURN;
	// endif

	PATCHBRANCH(branch2);
	mem(Stw, Rfp, Rreg, O(REG, FP));
	mem(Stw, Rlink, Rreg, O(REG, st));
	mem(Stw, RA0, Rreg, O(REG, s));
	CALL(rdestroy);

	loadi(Rreg, (ulong) &R);
	mem(Ldw, Rlink, Rreg, O(REG, st));
	mem(Ldw, Rfp, Rreg, O(REG, FP));
	mem(Ldw, Rmp, Rreg, O(REG, MP));

	// endif
	PATCHBRANCH(branch1);
```

```
        RETURN;
}

static void
macret(void)
{
        Inst i;
        ulong *branch1, *branch2, *branch3, *branch4, *branch5, *branch6;

        branch1 = code;
        BEQ(RA1, R0, 0);

        // if t(Rfp) != 0
        mem(Ldw, RA0, RA1, O(Type, destroy));
        branch2 = code;
        BEQ(RA0, R0, 0);

        // if destroy(t(fp)) != 0
        mem(Ldw, RA2, Rfp, O(Frame, fp));
        branch3 = code;
        BEQ(RA2, R0, 0);

        // if fp(Rfp) != 0
        mem(Ldw, RA3, Rfp, O(Frame, mr));
        branch4 = code;
        BEQ(RA3, R0, 0);

        // if mr(Rfp) != 0
        mem(Ldw, RA2, Rreg, O(REG, M));
        mem(Ldw, RA3, RA2, O(Heap, ref) - sizeof(Heap));
        ADDI(RA3, RA3, -1);

        branch5 = code;
        BEQ(RA3, R0, 0);

        // if --ref(arg) != 0
        mem(Stw, RA3, RA2, O(Heap, ref) - sizeof(Heap));
        mem(Ldw, RA1, Rfp, O(Frame, mr));
        mem(Stw, RA1, Rreg, O(REG, M));
        mem(Ldw, Rmp, RA1, O(Modlink, MP));
        mem(Stw, Rmp, Rreg, O(REG, MP));

        mem(Ldw, RA3, RA1, O(Modlink, compiled));
        branch6 = code;
        BEQ(RA3, R0, 0);

        // This part is a bit weird, because it should be the innermost
        // if-statement (in C terms), but the else of branch4 also ends up here.
        // This could be a mistake, but it's in at least the ARM and MIPS version.

        // if R.M->compiled != 0
        // if mr(Rfp) == 0
        PATCHBRANCH(branch4);
        JRL(RA0, 0);                            // Call destroy(t(fp))

        mem(Stw, Rfp, Rreg, O(REG, SP));        // R->SP = Rfp
        mem(Ldw, RA1, Rfp, O(Frame, lr));       // RA1 = Rfp->lr
        mem(Ldw, Rfp, Rfp, O(Frame, fp));       // Rfp = Rfp->fp
        mem(Stw, Rfp, Rreg, O(REG, FP));        // R->FP = Rfp

        JR(RA1, 0);                             // goto RA1, if compiled
        // does not continue past here

        // if R.M->compiled == 0
        PATCHBRANCH(branch6);
        JRL(RA0, 0);                            // Call destroy(t(fp))

        mem(Stw, Rfp, Rreg, O(REG, SP));        // R->SP = Rfp
        mem(Ldw, RA1, Rfp, O(Frame, lr));       // RA1 = Rfp->lr
        mem(Ldw, Rfp, Rfp, O(Frame, fp));       // Rfp = Rfp->fp
        mem(Stw, RA1, Rreg, O(REG, PC));        // R.PC = RA1
        mem(Ldw, Rlink, Rreg, O(REG, xpc));     // Rlink = R->xpc
        RETURN;                                 // return to xec uncompiled code

        // endif
        PATCHBRANCH(branch5);
```

```
	PATCHBRANCH(branch3);
	PATCHBRANCH(branch2);
	PATCHBRANCH(branch1);

	i.add = AXNON;
	punt(&i, TCHECK|NEWPC, optab[IRET]);
}

static void
macrelq(void)
{
	// Store frame pointer and link register, then return to xev
	mem(Stw, Rfp, Rreg, O(REG, FP));
	mem(Stw, Rlink, Rreg, O(REG, PC));
	mem(Ldw, Rlink, Rreg, O(REG, xpc));
	RETURN;
}

/* Main compilation functions */
static void
comi(Type *t)
{
	// Compile a type initializer
	int i, j, m, c;

	for (i = 0; i < t->np; i++) {
		c = t->map[i];
		j = i << 5;

		for (m = 0x80; m != 0; m >>= 1) {
			if (c & m)
				mem(Stw, Rh, RA2, j);

			j += sizeof(WORD*);
		}
	}

	RETURN;
}

static void
comd(Type *t)
{
	// Compile a type destructor
	int i, j, m, c;

	mem(Stw, Rlink, Rreg, O(REG, dt));

	for (i = 0; i < t->np; i++) {
		c = t->map[i];
		j = i << 5;

		for (m = 0x80; m != 0; m >>= 1) {
			if (c & m) {
				mem(Ldw, RA0, Rfp, j);
				CALL(base+macro[MacFRP]);
			}

			j += sizeof(WORD*);
		}
	}

	mem(Ldw, Rlink, Rreg, O(REG, dt));
	RETURN;
}

static void
typecom(Type *t)
{
	// Compile a type
	int n;
	ulong *tmp, *start;

	if (t == nil || t->initialize != 0)
		return;

```

```c
	tmp = mallocz(4096*sizeof(ulong), 0);
	if (tmp == nil)
		error(exNomem);

	codestart = tmp;
	codeend = tmp + 4096;
	iprint("Typecom np %d, size %d\n", t->np, t->size);
	code = tmp;
	comi(t);
	n = code - tmp;
	code = tmp;
	comd(t);
	n += code - tmp;
	free(tmp);

	n *= sizeof(*code);
	code = mallocz(n, 0);
	if (code == nil)
		return;

	codestart = code;
	codeend = code + n;

	start = code;
	t->initialize = code;
	comi(t);
	t->destroy = code;
	comd(t);

	segflush(start, n);

	if (cflag > 3)
		iprint("typ= %.8p %4d i %.8p d %.8p asm=%d\n",
			t, t->size, t->initialize, t->destroy, n);

	if (cflag > 6) {
		das(start, code-start);
	}
}

static void
patchex(Module *m, ulong *p)
{
	// Apply patches for a module. p is the patch array
	Handler *h;
	Except *e;

	for (h = m->htab; h != nil && h->etab != nil; h++) {
		h->pc1 = p[h->pc1];
		h->pc2 = p[h->pc2];

		for (e = h->etab; e->s != nil; e++)
			e->pc = p[e->pc];

		if (e->pc != -1)
			e->pc = p[e->pc];
	}
}

static void
commframe(Inst *i)
{
	// Compile a mframe instruction
	ulong *branch1, *branch2;
	loadi(R7, 0);

	op1(Ldw, i, RA0);
	branch1 = code;
	BEQ(RA0, Rh, 0);

	// if RA0 != H

	// RA3 = src->links[src2]->frame
	if ((i->add & ARM) == AXIMM) {
		mem(Ldw, RA3, RA0, OA(Modlink, links) + i->reg*sizeof(Modl) + O(Modl, frame));
	} else {
```

```
                // RA1 = src->links[src2]
                op2(Ldw, i, RA1);
                multiply(RA1, RA1, sizeof(Modl));
                ADD(RA1, RA1, RA0);

                // RA3 = src->links[src2]->frame
                mem(Ldw, RA3, RA1, O(Modl, frame));
        }

        mem(Ldw, RA1, RA3, O(Type, initialize));
        branch2 = code;
        BNEZ(RA1, 0);

        // if frame->initialize == 0
        op3(Laddr, i, RA0);
        // endif

        // if RA0 == H || frame->initialize == 0
        PATCHBRANCH(branch1);
        loadi(Rlink, RELPC(patch[i - mod->prog + 1]));
        loadi(R7, 7);
        CALLMAC(MacMFRA);

        // if frame->inititalize != 0
        PATCHBRANCH(branch2);
        loadi(R7, 8);
        CALLMAC(MacFRAM);
        op3(Stw, i, RA2);
}

static void
commcall(Inst *i)
{
        // Compile a mcall instruction
        ulong *branch;

        op1(Ldw, i, RA2);                       // RA2 = src1 = frame
        loadi(RA0, RELPC(patch[i - mod->prog+1])); // RA0 = pc
        mem(Stw, RA0, RA2, O(Frame, lr));       // frame.lr = RA0 = pc
        mem(Stw, Rfp, RA2, O(Frame, fp));       // frame.fp = fp
        mem(Ldw, RA3, Rreg, O(REG, M));         // RA3 = R.M
        mem(Stw, RA3, RA2, O(Frame, mr));       // frame.mr = RA3 = R.M

        op3(Ldw, i, RA3); // RA3 = src3 = Modlink

        branch = code;
        BEQ(RA3, Rh, 0);
        // If RA3 != H

        // RA0 = Modlink->links[src2]->pc
        if ((i->add&ARM) == AXIMM) {
                // i->reg contains the immediate of src2, don't have to store it in a register
                mem(Ldw, RA0, RA3, OA(Modlink, links) + i->reg*sizeof(Modl) + O(Modl, u.pc));
        } else {
                op2(Ldw, i, RA1);               // RA1 = src2

                // RA1 *= sizeof(Modl)
                multiply(RA1, RA1, sizeof(Modl));

                ADDI(RA1, RA1, RA3);
                mem(Ldw, RA0, RA1, OA(Modlink, links) + O(Modl, u.pc));
        }

        PATCHBRANCH(branch); // endif

        CALLMAC(MacMCAL);
}

static void
comcase(Inst *i, int w)
{
        // Compile a case instruction
        int l;
        WORD *t, *e;

        if (w != 0) {
```

```
			// Use the MacCASE macro
			op1(Ldw, i, RA1);
			op3(Laddr, i, RA3);
			CALLMAC(MacCASE);
	}

	// Get a pointer to the table
	t = (WORD*)(mod->origmp + i->d.ind+4);

	// Get the flag right before the table
	l = t[-1];

	/* have to take care not to relocate the same table twice -
	 * the limbo compiler can duplicate a case instruction
	 * during its folding phase
	 */

	if (pass == 0) {
		if (l >= 0)
			t[-1] = -l-1;	/* Mark it not done */
		return;
	}

	if (l >= 0) {			/* Check pass 2 done */
		return;
	}

	t[-1] = -l-1;			/* Set real count */
	e = t + t[-1]*3;

	while (t < e) {
		t[2] = RELPC(patch[t[2]]);
		t += 3;
	}

	t[0] = RELPC(patch[t[0]]);
}

static void
comcasel(Inst *i)
{
	// Same as comecase, but with double words
	int l;
	WORD *t, *e;

	t = (WORD*) (mod->origmp + i->d.ind + 8);
	l = t[-2];

	if (pass == 0) {
		if (l >= 0)
			t[-2] = -l-1;	/* Mark it not done */
		return;
	}

	if (l >= 0)			/* Check pass 2 done */
		return;

	t[-2] = -l-1;			/* Set real count */
	e = t + t[-2]*6;

	while (t < e) {
		t[4] = RELPC(patch[t[4]]);
		t += 6;
	}

	t[0] = RELPC(patch[t[0]]);
}

static void
comgoto(Inst *i)
{
	// Compile a goto instruction
	WORD *t, *e;

	op1(Ldw, i, RA1);		// RA1 = src
	op3(Laddr, i, RA0);		// RA0 = &dst
```

```
        SLLI(RA1, RA1, 2);              // RA1 = src*sizeof(int)
        ADD(RA1, RA1, RA0);             // RA1 += RA0
        mem(Ldw, RA0, RA1, 0);          // RA0 = dst[src]
        JR(RA0, 0);                     // goto dst[src]

        if (pass == 0)
                return;

        t = (WORD*)(mod->origmp+i->d.ind);
        e = t + t[-1];
        t[-1] = 0;

        while (t < e) {
                t[0] = RELPC(patch[t[0]]);
                t++;
        }
}

static void
comp(Inst *i)
{
        // Compile a single DIS instruction
        char buf[64];
        ulong *branch1, *branch2, *loop;

        switch (i->op) {
        default:
                snprint(buf, sizeof buf, "%s compile, no '%D'", mod->name, i);
                error(buf);
                break;
        case IMCALL:
                commcall(i);
                break;
        case ISEND:
        case IRECV:
        case IALT:
                punt(i, SRCOP|DSTOP|TCHECK|WRTPC, optab[i->op]);
                break;
        case ISPAWN:
                punt(i, SRCOP|DBRAN, optab[i->op]);
                break;
        case IBNEC:
        case IBEQC:
        case IBLTC:
        case IBLEC:
        case IBGTC:
        case IBGEC:
                punt(i, SRCOP|DBRAN|NEWPC|WRTPC, optab[i->op]);
                break;
        case ICASEC:
                comcase(i, 0);
                punt(i, SRCOP|DSTOP|NEWPC, optab[i->op]);
                break;
        case ICASEL:
                comcasel(i);
                punt(i, SRCOP|DSTOP|NEWPC, optab[i->op]);
                break;
        case IADDC:
        case IMULL:
        case IDIVL:
        case IMODL:
        case IMNEWZ:
        case ILSRW:
        case ILSRL:
                punt(i, SRCOP|DSTOP|THREOP, optab[i->op]);
                break;
        case IMODW:
                op1(Ldw, i, RA1);
                op2(Ldw, i, RA0);
                REM(RA0, RA0, RA1);
                op3(Stw, i, RA0);
                break;
        case IMODB:
                op1(Ldb, i, RA1);
                op2(Ldb, i, RA0);
                REM(RA0, RA0, RA1);
```

```
			op3(Stb, i, RA0);
			break;
		case IDIVW:
			op1(Ldw, i, RA1);
			op2(Ldw, i, RA0);
			DIV(RA0, RA0, RA1);
			op3(Stw, i, RA0);
			break;
		case IDIVB:
			op1(Ldb, i, RA1);
			op2(Ldb, i, RA0);
			DIV(RA0, RA0, RA1);
			op3(Stb, i, RA0);
			break;
		case ILOAD:
		case INEWA:
		case INEWAZ:
		case INEW:
		case INEWZ:
		case ISLICEA:
		case ISLICELA:
		case ICONSB:
		case ICONSW:
		case ICONSL:
		case ICONSF:
		case ICONSM:
		case ICONSMP:
		case ICONSP:
		case IMOVMP:
		case IHEADMP:
		case IHEADB:
		case IHEADW:
		case IHEADL:
		case IINSC:
		case ICVTAC:
		case ICVTCW:
		case ICVTWC:
		case ICVTLC:
		case ICVTCL:
		case ICVTFC:
		case ICVTCF:
		case ICVTRF:
		case ICVTFR:
		case ICVTWS:
		case ICVTSW:
		case IMSPAWN:
		case ICVTCA:
		case ISLICEC:
		case INBALT:
			punt(i, SRCOP|DSTOP, optab[i->op]);
			break;
		case INEWCM:
		case INEWCMP:
			punt(i, SRCOP|DSTOP|THREOP, optab[i->op]);
			break;
		case IMFRAME:
			commframe(i);
			break;
		case ICASE:
			comcase(i, 1);
			break;
		case IGOTO:
			comgoto(i);
			break;
		case IMOVF:
			op1(Ldd, i, F1);
			op3(Std, i, F1);
			break;
		case IMOVL:
			op1(Laddr, i, RA0);
			mem(Ldw, RA1, RA0, 0);
			mem(Ldw, RA2, RA0, 4);

			op3(Laddr, i, RA0);
			mem(Stw, RA1, RA0, 0);
			mem(Stw, RA2, RA0, 4);
			break;
		case IHEADM:
			punt(i, SRCOP|DSTOP, optab[i->op]);
			break;
			op1(Laddr, i, RA1);
			NOTNIL(RA1);

			if(OA(List, data) != 0) {
				ADDI(RA1, RA1, OA(List, data));
			}

			movmem(i);
			break;
		case IMOVM:
			punt(i, SRCOP|DSTOP|THREOP, optab[i->op]);
			break;
			op1(Laddr, i, RA1);
			movmem(i);
			break;
		case IFRAME:
			if(UXSRC(i->add) != SRC(AIMM)) {
				punt(i, SRCOP|DSTOP, optab[i->op]);
				break;
			}
			tinit[i->s.imm] = 1;
			loadi(RA3, (ulong) mod->type[i->s.imm]);
			CALL(base+macro[MacFRAM]);
			op3(Stw, i, RA2);
			break;
		case INEWCB:
		case INEWCW:
		case INEWCF:
		case INEWCP:
		case INEWCL:
			punt(i, DSTOP|THREOP, optab[i->op]);
			break;
		case IEXIT:
			punt(i, 0, optab[i->op]);
			break;
		case ICVTBW:
			op1(Ldbu, i, RA0);
			op3(Stw, i, RA0);
			break;
		case ICVTWB:
			op1(Ldw, i, RA0);
			op3(Stb, i, RA0);
			break;
		case ILEA:
			op1(Laddr, i, RA0);
			op3(Stw, i, RA0);
			break;
		case IMOVW:
			op1(Ldw, i, RA0);
			op3(Stw, i, RA0);
			break;
		case IMOVB:
			op1(Ldb, i, RA0);
			op3(Stb, i, RA0);
			break;
		case ITAIL:
			punt(i, SRCOP|DSTOP, optab[i->op]);
			break;
			op1(Ldw, i, RA0);
			NOTNIL(RA0);
			mem(Ldw, RA1, RA0, O(List, tail));
			movptr(i);
			break;
		case IMOVP:
			punt(i, SRCOP|DSTOP, optab[i->op]);
			break;
			op1(Ldw, i, RA1);
			NOTNIL(RA1);
			movptr(i);
			break;
		case IHEADP:
			punt(i, SRCOP|DSTOP, optab[i->op]);
```

```
                        break;
                        op1(Ldw, i, RA0);
                        NOTNIL(RA0);
                        mem(Ldw, OA(List, data), RA0, RA1);
                        movptr(i);
                        break;
                case ILENA:
                        punt(i, SRCOP|DSTOP, optab[i->op]);
                        break;
                        op1(Ldw, i, RA1);
                        MOV(RA0, R0);

                        branch1 = code;
                        BEQ(RA1, Rh, 0);

                        // if src != H
                        mem(Ldw, RA0, RA1, O(Array, len));
                        // endif

                        PATCHBRANCH(branch1);
                        op3(Stw, i, RA0);
                        break;
                case ILENC:
                        punt(i, SRCOP|DSTOP, optab[i->op]);
                        break;
                        op1(Ldw, i, RA1);
                        MOV(RA0, R0);

                        branch1 = code;
                        BEQ(RA1, Rh, 0);

                        // if RA1 != H
                        mem(Ldw, RA0, RA1, O(String, len));

                        branch2 = code;
                        BGE(RA0, 0, 0);

                        // if string->len < 0

                        // RA0 = abs(string->len)
                        NEG(RA0, RA0);

                        // endif

                        PATCHBRANCH(branch1);
                        PATCHBRANCH(branch2);
                        op3(Stw, i, RA0);
                        break;
                case ILENL:
                        punt(i, SRCOP|DSTOP, optab[i->op]);
                        break;
                        MOV(RA0, R0);                           // RA0 = 0
                        op1(Ldw, i, RA1);                       // RA1 = src

                        // while RA1 != H
                        loop = code;
                        BEQ(RA1, Rh, 0);

                        mem(Ldw, RA1, RA1, O(List, tail));      // RA0 = RA0->tail
                        ADDI(RA0, RA0, 1);                      // RA1++
                        JABS(loop);
                        // endwhile

                        PATCHBRANCH(loop);
                        op3(Stw, i, RA0);                       // return RA1
                        break;
                case ICALL:
                        op1(Ldw, i, RA0);
                        loadi(RA1, RELPC(patch[i - mod->prog + 1]));
                        mem(Stw, RA1, RA0, O(Frame, lr));
                        mem(Stw, Rfp, RA0, O(Frame, fp));
                        MOV(Rfp, RA0);
                        JDST(i);
                        break;
                case IJMP:
                        JDST(i);
                        break;
                case IBEQW:
                        branch(i, Ldw, EQ);
                        break;
                case IBNEW:
                        branch(i, Ldw, NE);
                        break;
                case IBLTW:
                        branch(i, Ldw, LT);
                        break;
                case IBLEW:
                        branch(i, Ldw, LE);
                        break;
                case IBGTW:
                        branch(i, Ldw, GT);
                        break;
                case IBGEW:
                        branch(i, Ldw, GE);
                        break;
                case IBEQB:
                        branch(i, Ldb, EQ);
                        break;
                case IBNEB:
                        branch(i, Ldb, NE);
                        break;
                case IBLTB:
                        branch(i, Ldb, LT);
                        break;
                case IBLEB:
                        branch(i, Ldb, LE);
                        break;
                case IBGTB:
                        branch(i, Ldb, GT);
                        break;
                case IBGEB:
                        branch(i, Ldb, GE);
                        break;
                case IBEQF:
                        branchfd(i, EQ);
                        break;
                case IBNEF:
                        branchfd(i, NE);
                        break;
                case IBLTF:
                        branchfd(i, LT);
                        break;
                case IBLEF:
                        branchfd(i, LE);
                        break;
                case IBGTF:
                        branchfd(i, GT);
                        break;
                case IBGEF:
                        branchfd(i, GE);
                        break;
                case IRET:
                        mem(Ldw, RA1, Rfp, O(Frame, t));
                        CALLMAC(MacRET);
                        break;
                case IMULW:
                        op1(Ldw, i, RA1);
                        op2(Ldw, i, RA0);
                        MUL(RA0, RA0, RA1);
                        op3(Stw, i, RA0);
                        break;
                case IMULB:
                        op1(Ldb, i, RA1);
                        op2(Ldb, i, RA0);
                        MUL(RA0, RA0, RA1);
                        op3(Stb, i, RA0);
                        break;
                case IORW:
                        op1(Ldw, i, RA1);
                        op2(Ldw, i, RA2);
                        OR(RA0, RA1, RA2);
                        op3(Stw, i, RA0);
                        break;
                case IANDW:
                        op1(Ldw, i, RA1);
                        op2(Ldw, i, RA2);
                        AND(RA0, RA1, RA2);
                        op3(Stw, i, RA0);
                        break;
                case IXORW:
                        op1(Ldw, i, RA1);
                        op2(Ldw, i, RA2);
                        XOR(RA0, RA1, RA2);
                        op3(Stw, i, RA0);
                        break;
                case ISUBW:
                        op1(Ldw, i, RA2);
                        op2(Ldw, i, RA1);
                        SUB(RA0, RA1, RA2);
                        op3(Stw, i, RA0);
                        break;
                case IADDW:
                        op1(Ldw, i, RA1);
                        op2(Ldw, i, RA2);
                        ADD(RA0, RA1, RA2);
                        op3(Stw, i, RA0);
                        break;
                case ISHRW:
                        op1(Ldw, i, RA1);
                        op2(Ldw, i, RA2);
                        SRL(RA0, RA2, RA1);     // Shift order is switched
                        op3(Stw, i, RA0);
                        break;
                case ISHLW:
                        op1(Ldw, i, RA1);
                        op2(Ldw, i, RA2);
                        SLL(RA0, RA2, RA1);     // Shift order is switched
                        op3(Stw, i, RA0);
                case IORB:
                        op1(Ldb, i, RA1);
                        op2(Ldb, i, RA2);
                        OR(RA0, RA1, RA2);
                        op3(Stb, i, RA0);
                        break;
                case IANDB:
                        op1(Ldb, i, RA1);
                        op2(Ldb, i, RA2);
                        AND(RA0, RA1, RA2);
                        op3(Stb, i, RA0);
                        break;
                case IXORB:
                        op1(Ldb, i, RA1);
                        op2(Ldb, i, RA2);
                        XOR(RA0, RA1, RA2);
                        op3(Stb, i, RA0);
                        break;
                case ISUBB:
                        op1(Ldb, i, RA1);
                        op2(Ldb, i, RA2);
                        SUB(RA0, RA1, RA2);
                        op3(Stb, i, RA0);
                        break;
                case IADDB:
                        op1(Ldb, i, RA1);
                        op2(Ldb, i, RA2);
                        ADD(RA0, RA1, RA2);
                        op3(Stb, i, RA0);
                        break;
                case ISHRB:
                        op1(Ldb, i, RA1);
                        op2(Ldb, i, RA2);
                        SRL(RA0, RA2, RA1);     // Shift order is switched
                        op3(Stb, i, RA0);
                case ISHLB:
                        op1(Ldb, i, RA1);
                        op2(Ldb, i, RA2);
                        SLL(RA0, RA2, RA1);     // Shift order is switched
                        op3(Stb, i, RA0);
```

```
	case IINDC:
		op1(Ldw, i, RA1);	// RA1 = src1 = string
		NOTNIL(RA1);

		op2(Ldw, i, RA2);	// RA2 = src2 = index

		mem(Ldw, RA0, RA1, O(String, len));	// RA0 = string->len

		if(bflag){
			MOV(RA3, RA0);
			branch1 = code;
			BGE(RA3, R0, 0);

			// if string->len < 0
			NEG(RA3, RA3);
			// endif

			PATCHBRANCH(branch1);
			BCK(RA2, RA3);
		}

		ADDI(RA1, RA1, O(String, data));

		branch2 = code;
		BGE(RA0, R0, 0);

		// if string->len < 0
		SLLI(RA2, RA2, 2);		// index = index << 2; in words, not bytes
		// endif

		PATCHBRANCH(branch2);
		mem(Ldw, RA3, RA1, RA2);	// RA3 = string[index]
		op3(Stw, i, RA3);
		break;
	case IINDL:
	case IINDF:
	case IINDW:
	case IINDB:
		op1(Ldw, i, RA1);				// RA1 = src1 = array
		NOTNIL(RA1);
		op3(Ldw, i, RA2);				// RA2 = src2 = index

		if(bflag) {
			mem(Ldw, RA3, RA1, O(Array, len));	// RA3 = array->len
			BCK(RA2, RA3);
		}

		mem(Ldw, RA1, RA1, O(Array, data));		// RA1 = array->data

		// Modify the index to match the data width
		switch(i->op) {
		case IINDL:
		case IINDF:
			SLLI(RA2, RA2, 3);
			break;
		case IINDW:
			SLLI(RA2, RA2, 2);
			break;
		}

		ADD(RA1, RA1, RA2);
		op2(Stw, i, RA1);
		break;
	case IINDX:
		op1(Ldw, i, RA1);			// RA1 = src1 = array
		NOTNIL(RA0);
		op3(Ldw, i, RA2);			// RA2 = src2 = index

		if(bflag){
			mem(Ldw, RA3, RA1, O(Array, len));	// RA3 = array->len
			BCK(RA2, RA3);
		}

		mem(Ldw, RA3, RA1, O(Array, t));	// RA3 = array->t
		mem(Ldw, RA3, RA3, O(Type, size));	// RA3 = array->t->size
		mem(Ldw, RA1, RA1, O(Array, data));	// RA1 = array->data
```

```

                MUL(RA2, RA2, RA3);                     // RA2 = index*size
                ADD(RA1, RA1, RA0);                     // RA1 = array->data + index*size
                op2(Stw, i, RA1);
                break;
        case IADDL:
        case ISUBL:
        case IORL:
        case IANDL:
        case IXORL:
                // The Dis instructions uses the format "src3 = src2 op src1",
                // which is opposite to RISC-V. To make the code more intuitive the order
                // is switched here, so the operations are "src3 = RA1.RA2 op RA3.RA4"

                // RA1, RA2 = src2
                op2(Laddr, i, RA0);
                mem(Ldw, RA1, RA0, 0);
                mem(Ldw, RA2, RA0, 4);

                // RA3, RA4 = src1
                op1(Laddr, i, RA0);
                mem(Ldw, RA3, RA0, 0);
                mem(Ldw, RA4, RA0, 4);

                switch (i->op) {
                case IADDL:
                        ADD(RA0, RA1, RA3);             // RA0 = src2[31:0] + src1[31:0]
                        ADD(RA2, RA2, RA4);             // RA2 = src2[63:32] + src1[63:32]

                        // Check for overflow
                        SLTU(RA1, RA0, RA1);            // RA1 = RA0 < src2[31:0] ? 1 : 0

                        // Add the overflow to the upper bits
                        ADD(RA2, RA2, RA1);

                        // Move the lower result to RA1
                        MOV(RA1, RA0);
                        break;
                case ISUBL:
                        SUB(RA0, RA1, RA3);             // RA0 = src2[31:0] - src1[31:0]
                        SUB(RA2, RA2, RA4);             // RA2 = src2[63:32] - src1[63:32]

                        // Check for underflow
                        SLTU(RA1, RA1, RA0);            // RA1 = src2[31:0] < RA0 ? 1 : 0

                        // Add the underflow to the upper bits
                        SUB(RA2, RA2, RA1);

                        // Move the lower result to RA1
                        MOV(RA1, RA0);
                        break;
                case IORL:
                        OR(RA1, RA1, RA3);
                        OR(RA2, RA2, RA4);
                        break;
                case IANDL:
                        AND(RA1, RA1, RA3);
                        AND(RA2, RA2, RA4);
                        break;
                case IXORL:
                        XOR(RA1, RA1, RA3);
                        XOR(RA2, RA2, RA4);
                        break;
                }

                // dst = RA1, RA2
                op3(Laddr, i, RA0);
                mem(Stw, RA1, RA0, 0);
                mem(Stw, RA2, RA0, 4);
                break;
        case ICVTWL:
                op1(Ldw, i, RA1);
                op2(Laddr, i, RA0);
                SRAI(RA2, RA1, 31);                     // Shift right 31 places to sign-extend
                mem(Stw, RA1, RA0, 0);
                mem(Stw, RA2, RA0, 4);
                break;
        case ICVTLW:
                op1(Ldw, i, RA0);
                op3(Stw, i, RA0);
                break;
        case IBEQL:
                branchl(i, EQ);
                break;
        case IBNEL:
                branchl(i, NE);
                break;
        case IBLEL:
                branchl(i, LE);
                break;
        case IBGTL:
                branchl(i, GT);
                break;
        case IBLTL:
                branchl(i, LT);
                break;
        case IBGEL:
                branchl(i, GE);
                break;
        case ICVTFL:
                ADDI(Rsp, Rsp, -16);

                op1(Ldd, i, F1);                // Load the double to convert
                op3(Laddr, i, Rarg);            // Load the destination as the first argument to _d2v

                // Round F1 by adding 0.5 or -0.5

                // F2 = 0.5
                LUI(Rta, SPLITH(&double05));
                mem(Ldd, F2, Rta, SPLITL(&double05));

                FSGNJD(F2, F2, F1);             // F2 = F1 >= 0 ? F2 : -F2
                FADDD(RM, F1, F1, F2);          // F1 += F2

                mem(Std, F1, Rsp, 8);           // Store F1 as the second argument, and call _d2v

                // Call _d2v
                mem(Stw, Rfp, Rreg, O(REG, FP));
                CALL(_d2v);
                loadi(Rreg, (ulong) &R);
                mem(Ldw, Rfp, Rreg, O(REG, FP));
                mem(Ldw, Rmp, Rreg, O(REG, MP));

                ADDI(Rsp, Rsp, 16);
                break;
        case ICVTLF:
                op1(Laddr, i, Rta);
                mem(Ldw, RA0, Rta, 0);
                mem(Ldw, RA1, Rta, 4);

                FCVTDWU(RM, F0, RA0);           // F0 = float(unsigned src[0:31])
                FCVTDW(RM, F1, RA1);            // F1 = float(src[32:63])

                // F2 = 4294967296
                LUI(Rta, SPLITH(&double4294967296));
                mem(Ldd, F2, Rta, SPLITL(&double4294967296));

                FMADDD(RM, F0, F1, F2, F0);     // F0 = F1 * F2 + F0

                // Store the result
                op3(Std, i, F0);
                break;
        case IDIVF:
                op1(Ldd, i, F1);
                op2(Ldd, i, F2);
                FDIVD(RM, F1, F2, F1);
                op3(Std, i, F1);
                break;
        case IMULF:
                op1(Ldd, i, F1);
                op2(Ldd, i, F2);
                FMULD(RM, F1, F2, F1);
                op3(Std, i, F1);
                break;
        case ISUBF:
                op1(Ldd, i, F1);
                op2(Ldd, i, F2);
                FSUBD(RM, F1, F2, F1);
                op3(Std, i, F1);
                break;
        case IADDF:
                op1(Ldd, i, F1);
                op2(Ldd, i, F2);
                FADDD(RM, F1, F2, F1);
                op3(Std, i, F1);
                break;
        case INEGF:
                op1(Ldd, i, F1);
                FSGNJND(F1, F1, F1);
                op3(Std, i, F1);
                break;
        case ICVTWF:
                op1(Ldw, i, RA0);
                FCVTDW(RM, F1, RA0);
                op3(Std, i, F1);
                break;
        case ICVTFW:
                op1(Ldd, i, F1);
                FCVTWD(RM, RA0, F1);
                op3(Stw, i, RA0);
                break;
        case ISHLL:
                /* should do better */
                punt(i, SRCOP|DSTOP|THREOP, optab[i->op]);
                break;
        case ISHRL:
                /* should do better */
                punt(i, SRCOP|DSTOP|THREOP, optab[i->op]);
                break;
        case IRAISE:
                punt(i, SRCOP|WRTPC|NEWPC, optab[i->op]);
                break;
        case IMULX:
        case IDIVX:
        case ICVTXX:
        case IMULX0:
        case IDIVX0:
        case ICVTXX0:
        case IMULX1:
        case IDIVX1:
        case ICVTXX1:
        case ICVTFX:
        case ICVTXF:
        case IEXPW:
        case IEXPL:
        case IEXPF:
                punt(i, SRCOP|DSTOP|THREOP, optab[i->op]);
                break;
        case ISELF:
                punt(i, DSTOP, optab[i->op]);
                break;
        }
}

static void
preamble(void)
{
        if(comvec)
                return;

        comvec = malloc(20 * sizeof(*code));
        if(comvec == nil)
                error(exNomem);
        code = (ulong*)comvec;
        codestart = code;
        codeend = code + 10;

        loadi(Rh, (ulong) H);
```

```
	loadi(Rreg, (ulong) &R);
	mem(Stw, Rlink, Rreg, O(REG, xpc));
	mem(Ldw, Rfp,   Rreg, O(REG, FP));
	mem(Ldw, Rmp,   Rreg, O(REG, MP));
	mem(Ldw, RA0,   Rreg, O(REG, PC));
	JR(RA0, 0);

	if (cflag > 4) {
		iprint("preamble\n");
		das(codestart, code-codestart);
	}

	segflush(comvec, ((ulong)code-(ulong)comvec) * sizeof(*code));
}

int
compile(Module *m, int size, Modlink *ml)
{
	Link *l;
	Modl *e;
	int i, n;
	ulong *s, *tmp;

	iprint("compile\n");

	base = nil;
	patch = mallocz(size*sizeof(*patch), 0);
	tinit = malloc(m->ntype*sizeof(*tinit));
	tmp = mallocz(2048*sizeof(ulong), 0);

	if (patch == nil || tinit == nil || tmp == nil)
		goto bad;

	// Set base so that addresses are at the same order of magnitude in both passes
	base = tmp;

	preamble();
	codestart = tmp;
	codeend = tmp + 2048;

	mod = m;
	n = 0;
	pass = 0;
	nlit = 0;

	// Do the first pass
	for (i = 0; i < size; i++) {
		codeoff = n;
		code = tmp;
		comp(&m->prog[i]);
		patch[i] = n;
		n += code - tmp;
	}
	iprint("first pass used %d instructions\n", n);

	// Generate macros at the end
	for (i = 0; i < NMACRO; i++) {
		codeoff = n;
		code = tmp;
		mactab[i].gen();
		macro[mactab[i].idx] = n;
		n += code - tmp;
	}

	iprint("first pass and macros used %d instructions\n", n);

	free(tmp);
	base = mallocz((n+nlit)*sizeof(*code), 0);
	codestart = base;
	codeend = base + n + nlit;
	if (base == nil)
		goto bad;

	iprint("base address: 0x%p\n", base);
	iprint("mod->prog:    0x%p\n", mod->prog);
	iprint("size:         %d\n", size);

	if (cflag > 3)
		iprint("dis=%5d %5d risc-v=%5d asm=%.8p: %s\n",
			size, size*sizeof(Inst), n, base, m->name);

	// Prepare for the next pass
	pass++;
	nlit = 0;
	litpool = base + n;
	code = base;
	n = 0;
	codeoff = 0;

	// Translate the instructions
	iprint("compile second pass\n");
	for (i = 0; i < size; i++) {
		s = code;
		comp(&m->prog[i]);

		if (patch[i] != n) {
			// The previous instruction used a different number of instructions
			// than in the first pass, messing up the offsets
			if (cflag <= 4)
			    iprint("%3d %D\n", i, &m->prog[i-1]);
			iprint("First and second pass instruction count doesn't match\n");
			iprint("first pass: %lud\nsecond pass: %d\n", patch[i], n);
			urk("phase error");
		}

		if (cflag > 4) {
			iprint("%3d %D\n", i, &m->prog[i]);
			das(s, code-s);
		}

		n += code - s;
	}

	// Insert the macros
	iprint("compile second macro\n");
	for (i = 0; i < NMACRO; i++) {
		s = code;
		mactab[i].gen();

		if (macro[mactab[i].idx] != n) {
			iprint("mac phase err: %lud != %d\n", macro[mactab[i].idx], n);
			urk("phase error");
		}

		n += code - s;

		if (cflag > 5) {
			iprint("%s:\n", mactab[i].name);
			das(s, code-s);
		}
	}

	iprint("compile m->ext types\n");
	for (l = m->ext; l->name; l++) {
		l->u.pc = (Inst*) RELPC(patch[l->u.pc - m->prog]);
		typecom(l->frame);
	}

	if (ml != nil) {
		e = &ml->links[0];

		iprint("compile ml->links types\n");
		for (i = 0; i < ml->nlinks; i++) {
			e->u.pc = (Inst*) RELPC(patch[e->u.pc - m->prog]);
			typecom(e->frame);
			e++;
		}
	}

	iprint("compile m->type types\n");
	for (i = 0; i < m->ntype; i++) {
		if (tinit[i] != 0)
```

```
				typecom(m->type[i]);
	}

	iprint("compile patches\n");
	patchex(m, patch);
	m->entry = (Inst*) RELPC(patch[mod->entry - mod->prog]);

	iprint("compile done\n");
	free(patch);
	free(tinit);
	free(m->prog);
	m->prog = (Inst*) base;
	m->compiled = 1;
	segflush(base, n*sizeof(*base));
	return 1;

bad:
	iprint("compile failed\n");
	free(patch);
	free(tinit);
	free(base);
	free(tmp);
	return 0;
}
```