Inventory of Machine-Dependent Material

David A. Holland

V3, March 7, 2017[*]

1 Preliminaries

This is a collection and also analysis of the machine-dependent material found in an operating system. I have gone through first the OS/161 tree (as it is simple and contains almost all core functionality) and then the NetBSD tree (which is not simple but also is known to support a wide variety of target platforms) and catalogued the material. Note that it is an inventory of what I found in a production system and not an inventory of what we need for this project. (OS/161 is an instructional OS we use for teaching.)

For the purposes of at least this discussion I'm drawing the following distinction:

- The *machine* is the CPU architecture.
- The *platform* is the system board architecture.

This is based on the practical observation that at least historically the same processor chips have been used in many often fundamentally different system boards. And, at least theoretically, a system board architecture can be used with different processor types; although in practice this is rare. It is an approximate distinction, especially historically; e.g. back in the Sun M68K days, the MMU architecture was technically a property of the system board, and today some issues (such as cache configuration, or locating additional processor cores) span both the CPU and the system board. However, it is nonetheless a useful distinction.

In OS/161 the *machine* is `mips` and the *platform* is `sys161`. In NetBSD... things are a bit more complicated. Notions of machines and platforms in the real world are complicated, because sometimes there's a 1-1 mapping between them and sometimes there isn't, and also because vendors of architectures and platforms rarely ship designs that evolve in an orderly fashion. Roughly speaking NetBSD has one "port" (separate build in the build system, and separate set of precompiled install binaries) for every distinct pair of system board architecture and processor type; but the long-term goal is to move to one "port" per processor type and just ship different kernels for different boards. (This is complicated by the fact that some processor types also need multiple builds based on different ABI definitions, and other complications and lossage introduced by vendor issues.) There isn't any one thing that's quite the same as what I'm calling "platform"; however, the NetBSD concept `MACHINE_ARCH` is roughly comparable to what I'm calling *machine*.

After going around multiple times I decided the best way to organize this was to list the topics twice, first grouped from an OS perspective and with minimal discussion (just enough material to clearly identify each item), and then grouped by what the code generation requirements appear to be, with discussion as needed.

Between these lists I discuss some background and shared material pertinent to the code generation requirements.

[*]An earlier version dated February 23, 2016 may have circulated - that was an internal review draft. This version has no technical changes but has been edited for comprehensibility and context.

2 OS-perspective inventory

This inventory is divided along classical OS lines: kernel-level material vs. user-level material, with some things that are shared between both ("common") split out. The kernel-level material is organized by kernel subsystem. (The decomposition of kernel subsystems is based on my prior work – as I was never able to publish most of that, if anyone reads this who isn't familiar with it has questions or commentary on it please contact me – hopefully the categories are self-explanatory. Why the categories are the way they are isn't immediately pertinent.)

I've also moved bootloader and compiler/toolchain material to their own categories as these are materially different from the others.

I tried dividing these lists up into *machine*-dependent and *platform*-dependent sections; this was not very effective, so I backed it out.

2.1 Common material

- **Various standard type and type-size declarations.** This includes for example the proper definition of `size_t`.
- **Standard or semi-standard MD header files.** The best example of this is the MIPS `regdefs.h`, which contains cpp macros with symbolic names for the registers. (This material conventionally isn't built into the assembler.)
- **Endianness.** This includes both the declaration of endianness and also implementations for endianness-related functions like `ntohl` and `htobe64`.
- **`setjmp`.** Implementation of `setjmp`, `longjmp`, and the `jmp_buf` type. (This might or might not be wanted in the kernel, and the kernel version might be different depending on the floating-point ABI...)
- **Memory barrier operations.** Both kernel and userland need to be able to issue memory barriers from MI code. OS/161 has an API for this we can use. NetBSD has a different (uglier) one.
- **Atomic operations.** Both kernel and userland need to be able to use some MI atomic operations interface. NetBSD has an API for this we can use, although it's not free of issues. We should maybe also consider using the interface from C11, if it's adequate and we have a compiler that supports it.
- **Goop for `float.h` and `fenv.h`.** This stuff is annoying but required for userland in order to be able to build standards-compliant software.
- **Workarounds for CPU bugs.** Most CPUs have at least some bugs (e.g. the infamous Pentium `FDIV` and `f00f` bugs) and some have large numbers of glaring problems (e.g. some MIPS models); for many there are recommended workarounds that involve doing this or that special thing at this or that time.
- **Signal frames.** Declarations for what gets pushed on the user stack when a signal is delivered, plus any trampoline code needed to handle it. (The latter is "common" not because the kernel takes signals, but because it might be necessary for trampoline code to be linked into the kernel and mapped into userspace.)
- **Register set declarations for `ptrace`.** A necessary part of debugging another process is inspecting its register state, which is inherently MD. The declarations need to be shared; also in practice they may need to agree with declarations used by gdb (or lldb), since presumably we aren't writing a debugger and at least gdb comes with its own preconceived ideas.
- **Register set declarations for ELF coredumps.** (And also crashdumps.) These might or might not be the same as the `ptrace` register sets, but probably also

need to agree with gdb's preconceived notions.

- **ELF relocation codes.** The codes and semantics of relocation types in executable/binary files are machine-dependent. Both the kernel and userspace potentially need to know about these; in particular a kernel module loader usually needs to handle at least some relocations.

- **Any `asm.h` header for assembly code.** Most of the hand-written assembler in NetBSD (not so much in OS/161, but OS/161 has much less assembler in it) uses a collection of semiuniform macros for things like marking symbols external. Given that we're attempting to just generate all the assembly code, generating such a header (at least in a compatible fashion) may not be useful; on the other hand, this remains to be seen and it might be.

2.2 Kernel-level material

2.2.1 `locore`

This category covers exception handling and trap logic.

- **Exception entry code.** The assembler code that processor-level traps jump to.

- **Trap dispatching code.** The higher-level code that interprets processor-level trap codes and takes kernel actions based on them.

- **Signal posting code.** The code that manipulates trap frames to cause signals to be delivered on return to userspace.

- **FPU context save/restore.** On most platforms FPU context switching is (or should be) done lazily when needed, so it's separate from other processor context manipulation.

- **Floating point emulation.** On some platforms some floating point instructions always trap, and the kernel's responsible for doing the work. Probably we don't need to care about this.

- **Busywait timing loop.** In real life there is stupid hardware, and some stupid hardware has timing requirements, and often these requirements lead drivers to need short uninterruptible timing loops. (If we're doing x86 we need this.)

- **Logic for coping with pipeline hazards.** This is not a piece of code so much as a crosscutting concern, but it affects mostly code in locore.

- **Kernel profiling hooks.** Kernel profiling generally requires some MD logic. We might or might not care about being able to do kernel profiling.

- **Code for handling cycle counters.** Self-explanatory.

2.2.2 `thread`

This category covers kernel-level (in-kernel) threads, and also CPUs/cores and CPU handling.

- **How to handle `curthread`/`curcpu`.** The best way to do this depends on properties of the processor architecture. Ideally we'd be able to pick this automatically rather than have someone specify it.

- **Spinlocks.** Most of a spinlock implementation is MI, but the choice of exactly which instructions to use is not. It isn't always the case that one should just use the atomic ops library.

- **Code for on-chip timers.** Many recent CPUs have on-chip timers meant for timeslicing and other low-latency use. These need to be manipulated. Treating them as devices is not necessarily a good idea as the interfaces that involves tend to be heavyweight. They aren't purely registers in the RTL sense either as their state changes spontaneously rather than in response to instruction execution.

- **How many interrupt levels to use.** Traditionally BSD kernels have many interrupt levels; recent thinking tends to be that there should be fewer. The set of *potential* interrupt levels is a property of the MI kernel code; which of these are actually distinct is not.

- **Interrupt on/off code.** Generally processors have at least one on-chip interrupt on/off switch; some processors have multiple layers or on-chip support for interrupt priorities. (This is connected to but distinct from bus-level interrupt masking, below.)

- **Interrupt wait/suspend.** One of the things one needs to be able to do is idle the current CPU until an interrupt comes in.

- **CPU halt.** Another thing one needs to be able to do is halt the current CPU.

- **Thread switch code.** The kernel-level thread switch code.

- **Thread start code.** The kernel-level assembler code that starts up a new thread.

- **CPU identification logic.** CPUs have feature bits and model codes and all manner of related twaddle, and it's usually at least necessary to detect and print; sometimes the results are important for dealing with other things too.

- **Maximum number of cpus.** For sanity reasons, code that deals with groups of CPUs and such assumes there's some maximum number. This maximum number is normally per-platform rather than per-machine (in the senses above) as it depends more on things like the maximum number of sockets the motherboard architecture allows than it does on anything in the CPU itself.

- **How you find secondary CPUs and cores.** Some of this is platform-dependent and follows from bus and configuration logic; for on-chip cores, at least for some processors, it may not be.

2.2.3 `vm`

This category covers the virtual memory system.

- **VM-related constants.** There's a number of more or less standard constants whose values are machine-dependent; this includes things like `PAGE_SIZE` and also memory layout properties like `USERSTACK`.

- **Sizes and usage restrictions of superpages, if any.** Most modern architectures have some concept of superpages, but the concepts and implementations vary widely.

- **Detecting RAM size.** One has to find out somehow how much memory there is, and also where it lives. Probing usually isn't particularly safe. The means for doing this are (at least normally) platform-specific rather than machine-specific.

- **Properties of different regions of RAM.** In the real world there are issues such that some regions of RAM are better or different; e.g. on many platforms there are bus-level restrictions on addresses such that DMA can be done only to some regions of memory. On some platforms, different regions of RAM are faster or slower for backend reasons. And on NUMA platforms, memory regions are bound to cores and CPUs.

- **`copyin`/`copyout`.** This family of functions transfers data safely between kernel and user space. The implementation depends heavily on the processor design. Note that while for many processors there's a *tidy* implementation based on `setjmp` and `longjmp`, this implementation is not very efficient... and on some processors, or some configurations of some processors, it won't work at all. (Among other things, it relies on the current userspace being mapped into kernel space; current thinking about security precautions and risks is that this is undesirable.) There are also silly processors where one needs to use special instructions.

- **Cache control logic.** This includes both the instruction-level actions for manipulating caches, and the next level up that issues cache writebacks and invalidations needed to support DMA... and also to cope with split I/D caches, VIPT caches, and other follies that the hardware world likes to foist on us. (Open question: do we try to support VIVT caches so we can use various old hardware as test cases, or rule this out as too horrifying?)
- **VM system code and headers.** The VM system itself, or at least the MD part of it. This includes at least what's classically found in a BSD pmap module and also any further low-level MMU manipulation logic.

2.2.4 `dev`

This category covers devices, buses, and interrupts, but does not include device or bus drivers per se, as drivers for most peripheral hardware and hardware chipsets are (rightly) considered MI. Essentially everything in this category is platform-dependent rather than machine-dependent.

- **`bus_machdep.c` for various buses.** While the code for e.g. PCI itself is MI, the MI code necessarily interfaces to an MD module that knows at the lowest level where registers and interrupts are. We also have to figure out which buses we need this logic for.
- **`bus_space`, `bus_dma` and similar material.** NetBSD has an abstract bus framework, which has proven very useful as this allows a lot more driver logic to be MI than otherwise. However, it has non-MI bits that need to be provided.
- **Interrupt identification and dispatching code.** When the trap handling code determines that what it's got is an interrupt, other code needs to figure out whose interrupt it is and send it off to the right device or bus driver. This category also includes any needed logic for wiring interrupts to CPUs.
- **Bus-level interrupt masking code.** Similarly, sometimes it's necessary to mask specific interrupts, e.g. from bus slots with nothing in them.
- **Early console.** Before the real device driver for the system console probes and attaches, we still potentially want to be able to print messages to the screen. Usually one does this via firmware, or sometimes via an extra hokey semi-driver that does the kinds of things such firmware would do. In OS/161 there is no such thing, but that's a corner one can't really cut in real life.
- **Early disk I/O.** (Maybe.) It might also be necessary to know how to do disk I/O via firmware. Certainly any bootloader would need to know about this (but see below); it might also be necessary to load drivers or whatever, depending on aspects of the MI kernel architecture we only get to choose if we write our own kernel.
- **Interface to firmware.** For these operations, and also typically for power management functions and other goop, and sometimes for probing devices, one wants to be able to interact with system firmware.
- **Drivers for any platform-specific hardware.** Some platforms have unique hardware for which no MI driver exists and where writing one doesn't really make sense. Various examples exist, mostly legacy of one form or another.

2.2.5 `syscall`

This category covers system calls and system call handling.

- **System call argument collection.** While the system call table per se should be machine-independent (though it isn't in Linux, along with many other things demonstrating poor design) one needs machine-dependent logic to collect system call arguments into some form that the MI system call entry points in the kernel can accept. What this entails depends on the system call ABI and also to some extent on how the MI system call material is structured.
- **Trapframe updating code for syscall return.** Relatedly, one needs to update the userlevel processor state when returning from a system call.
- **Trapframe updating code for fork return.** One also needs to update the userlevel processor state when entering userlevel in a newly forked child process.
- **ELF machine type and word-size codes.** Assuming we don't invent a new format for program binaries, we need to know the proper code numbers for binaries we should attempt to execute.
- **`ptrace` register handling.** One of the operations in `ptrace` is to retrieve the register state of the target process. This is inherently MD. Obviously we don't need `ptrace` if we don't bother supporting a debugger; see below for discussion of that.
- **`ptrace` single-step handling.** Some processors have a single-step mode, which debuggers will expect to be made available through `ptrace`.
- **Glue for foreign OS system call compatibility.** This means the machine-level material needed for e.g. running Linux binaries on NetBSD, which more or less means extra different copies of the above material in this category. We don't need to care about this.
- **Native 32-on-64 compatibility.** Similarly, one needs another set of logic for running 32-bit binaries on a 64-bit machine. On some platforms with multiple ABIs there are more combinations, too; e.g. a 64-bit MIPS kernel theoretically ought to be able to run N64, O64, N32, and O32 binaries in both native and opposite endianness. We almost certainly don't need to care about this.
- **Machine-dependent system calls.** For almost every processor type there's at least one or two system calls, and often they more or less need to exist for 3rd-party software to run. E.g. on mips there's `mips_cachectl`; on x86 there's `i386_iopl`, and so on. Note that we need a JVM and the JVM's JIT is bound to need `mips_cachectl`...

2.2.6 `main`

This category covers startup and initialization code.

- **Kernel startup code.** This is the assembly code that runs between the time the bootloader invokes the kernel and execution reaches the kernel's machine-independent `main()`.

2.2.7 `conf` and `mk`

This category covers the system configuration and build environment.

- **Compiler and linker flags.** On some platforms some of the other considerations above may lead to needing machine-specific compiler and/or linker flags. For example, one might want to reserve a register to hold `curthread`. Also on some platforms kernels should be built with non-default flags;

e.g. on MIPS you generally want to disable PIC.

- **Linker script.** It is almost always necessary to provide a custom linker script to link a kernel, and the details are machine-dependent and sometimes platform-dependent as well.
- **What to put in the kernel config.** A kernel config contains buses and devices and also other things (file systems, network widgets, ...) and somehow we need to know what is and isn't relevant.

2.2.8 Other bits

- **MD hooks for big in-kernel things.** NetBSD has a number of fairly big-ticket items in the kernel that require some amount of MD code. Examples include `ddb` (the historical in-kernel debugger), `kgdb` (hooks for remote gdb into the kernel), and `sljit` (the JIT for BPF). Most of these we probably don't need, although some kernel debugging support is probably desirable. (See discussion below.)
- **Kernel module loader.** A full-featured kernel is capable of loading modules into the kernel on the fly; portions of the module loader involve relocations and are necessarily MD. We don't actually need this for this project. (Provided we don't end up stuck using Linux, where the module loader isn't optional.)

2.3 User-level Material

2.3.1 In core libraries

- **Any user-level manifestations of cache control logic.** Most of the things that require cache manipulation happen only in the kernel; but e.g. on MIPS, which has split I/D caches, any kind of dynamic code generation or dynamic code loading requires cache manipulation.
- **MD bits of libm and libc floating-point things.** Basically, all the stuff that supports using the FPU.
- **Softfloat code.** For some platforms you need or want a software floating point library. Portions of this are inherently machine-dependent. For this project we probably don't care, but it depends on what range of platforms we want to support and in particular on what vintage platforms we want to be able to use as test cases for expressivity.
- **Profiling logic.** Like kernel profiling, user profiling requires some amount of machine-dependent logic.
- **System call entry stubs.** This is the assembler code that actually invokes each system call.
- **Startup code.** This is the logic that sits between user program startup and the invocation of `main`, conventionally known as `crt0` or "crtstuff". Much, but not all, of it is MI.
- **ELF dynamic linker.** Fairly significant parts of the ELF dynamic linker are machine-dependent.

2.3.2 In libpthread

- **Thread switch code for userlevel threads.** If the thread library multiplexes user contexts on system contexts, it needs to do context switches. (It might do this by making system calls, but if so those system calls will contain similar MD logic.) NetBSD's current thread library does not multiplex in this fashion, but others may.
- **Thread start code for userlevel threads.** Like in the kernel, to start up a thread you typically need a small amount of MD assembler to initialize registers.

- **Further MD bits of libpthread.** I am not entirely up on what else in libpthread is machine-dependent, and I haven't looked all that closely yet because pthread implementations tend to be messy.

2.3.3 Elsewhere

- **Platform-dependent parts of installer.** NetBSD's installer has per-platform logic for things like choosing the right kind of partition tables for the disks, and knowing what kind of disk devices are expected to appear, and so on.

- **MD bits of `cpuctl`.** In NetBSD there's a `cpuctl` binary that can be used for turning CPUs on and off and doing other things with them, some of which are MD. We might or might not need to actually implement this in practice.

- **MD bits of `libkvm`.** This is basically code for reading pagetables, so it should be a small addendum to the VM system logic.

There is also in some places (e.g. openssl) assembler code that's there because someone thinks (rightly or wrongly) that handwritten assembler will be faster than compiler output. In our environment all or nearly all of this can and should just be disabled.

2.4 Bootloader

We are not doing an x86 bootloader (not unless by the end of the project it looks within reach) as x86 bootloaders require many stages and both have very tight constraints and require doing horrible things like switching back and forth between protected mode and real mode. Other platforms' bootloader situations are potentially much more tractable, though. Therefore, the relevance of the following bits is unclear.

- **Bootloader.** The bootloader itself is necessarily platform-specific as it needs to know how to talk to firmware to use the console and read blocks from disk. Also it has to know which blocks to read from disk, and how to interpret them; and on some platforms (e.g. x86) it has to do a series of mode changes. Often bootloaders need to be multiple-stage, and then they often need to know different ways to do these things depending on which stage they're in. And so on.

- **Bootloader install tool.** Installing a bootloader is also usually platform-specific, although typically somewhat less so than the bootloader code itself.

- **Kernel image conversion tools.** On some systems the firmware knows how to load kernels... or at least, bootloader images that it thinks are kernels. But often the images in question have to be in some legacy format like ECOFF or SunOS 4.x `a.out`, and so NetBSD includes a range of tools for doing these conversions, most or all of which have at least a minimal MD component.

2.5 Compiler and toolchain

- **Binutils.** This means the things in GNU binutils; an assembler and linker, and also the other more or less trivial tools like `ar` and `nm` and `ranlib`.

- **Compiler.** Yeah, the compiler. Unless we decide to write a new OS in some other language, we need a C compiler.

- **Debugger.** In theory we could get away without a debugger: any debugging we need to do during development using existing platforms can be done with an existing debugger using its existing MD backend, and in theory once we're done, we can reconfigure for a future platform without needing to debug anything. In practice this isn't a great idea; even if everything we do is perfect, it's still reasonable to assume that in the future someone might write a new module of some kind and make a mistake and want to debug it. So we should really come up with some story for debugging.

- **JVM.** For this project we need to host a JVM. Until proven otherwise we had best assume that the JVM will need a JIT in order to perform adequately. Another option might be compiling Java to native machine code, as gcc can already do – this affects the architecture of the Java code and therefore isn't free but might be possible.
- **Lint or other program analyzers.** NetBSD still ships and sort of maintains BSD lint, which requires a few MD declarations. At this point its value is questionable and we could punt it; however, having something of the sort (e.g. sparse or splint) is probably worthwhile, and the cost is probably low. (Remember that like a compiler these need to know about the target platform even if they run purely on the build platform.)

3 Background discussion

Some points of discussion, prior to wading into my specific thoughts on each item.

3.1 Register transfer lists

The basic compiler technology for specifying and reasoning about instructions (which are inherently machine-dependent) is the idea of a register transfer list: given some description of CPU state, for each instruction provide a register transfer list: for each element of CPU state affected this specifies the output value as a function of the input state. These specifications provide the semantic meaning of each instruction; this makes it possible for a code generator to reason about the cumulative meaning of sequences of instructions.

Since raw RTLs are large and cumbersome, one of Norman Ramsey's projects (λ-RTL) is a tool for reading a more palatable and less verbose source form, grinding it, and generating the complete form as output. This is certainly a concept we can use; whether we can use the existing λ-RTL implementation or need a new one depends on what we end up needing in our RTLs and how this interacts with assumptions made by the existing code.

Note that since compilers do not work with the privileged and special-purpose instructions we need to be able to handle (for things like MMU and cache control, for dealing with exceptions, and so forth) at a minimum the RTLs we'll need will have a lot more stuff in them, because of all the privileged processor state, and some of the instruction descriptions will be horrifically complicated compared to anything a compiler normally handles. These issues by themselves shouldn't pose expressivity problems, though they might reveal scaling problems in preexisting implementations.

I am not at the moment clear on how one specifies calling conventions in this environment. (I suspect that one doesn't, and there's additional stuff in Norman's later work for handling this.) We will need to cope with calling conventions (for system call arguments, and also for thread switching) but I don't foresee anything materially different from what a compiler needs to handle; and therefore, these issues are, though not necessarily simple, at least tractable.

There are, however, other things for which pure RTLs aren't going to be sufficient. The first of these is processor modes: while the processor mode is a piece of state and can be handled like any other piece of state, that probably isn't adequate. At a minimum we will need to be able to use processor modes as predicates in transfer lists; that is maybe ok, but we'll also need to be able to specify prerequisites for entering particular processor modes. In particular the long startup sequence of an x86 goes through multiple modes and requires many steps, and if we want to be able to synthesize this we'll need to be able to treat these modes as goals and search for sequences of instructions that result in all the prerequisites being satisfied. We should be able to use a theorem prover for the search; but we need the description language to be able to characterize what we need to search for.

Another thing that concerns me is that on some processors, most notably x86, pieces of the processor state sit in main memory. For example, there's a thing called the Interrupt Descriptor Table (IDT) that among other things holds the entry point addresses for machine-level traps. This lives in main memory; its *address* is loaded into the processor with the `lidt` instruction. One can keep track of its address easily enough; but to model what happens on an exception, one needs to keep track of the contents of the memory it points to. And at least in a naive world that means the RTL specification for the x86 has to track the state of the entire system memory, because any access to main memory might potentially update the IDT or update some chunk of memory that is *later* used as the IDT. That is unlikely to be workable; for the time being I am not sure what to do about it. (Note that there are four or five of these things in the processor, too, not just one, and at least one of them *is* routinely updated on the fly.)

Another issue is the state of caches. While compilers do nowadays model caches, I don't think they typically do it with RTLs, and they do it with a different goal: modeling latency. We need to model it to know when we need to do invalidations and writebacks. Trying to model the complete cache state in RTL would require a complete specification of the cache logic, which isn't remotely feasible for a wide variety of reasons. Instead for the kind of cache operations we need we likely want some kind of probabilistic model that matches the way we reason about caches when coding by hand: these addresses may be in the cache and that's inconsistent with what we need to do next so we need to do a flush.

This also doesn't address dealing with the differences between PIPT, VIPT, and VIVT caches, which I would really like to handle with a systematic model as there are too many ways to get it wrong otherwise.

It seems to me that RTLs can't really encode timers, so it isn't clear how one deals with them.

And of course, to handle the VM system requires representing the way a mapping is specified, which is quite different from pure processor state. This will require some other specification technology yet to be determined.

4 Code generation inventory

Easy stuff first.

4.1 Trivial.

The following things follow directly from very basic properties of the processor architecture.

- **Various standard type and type-size declarations.** For modern processors there are basically two sets of values, one for 32-bit processors and one for 64-bit processors. Even if we want to support 36-bit processors and other exotica, it's not that different or that difficult.

- **Any `asm.h` header for assembly code.** If we want one of these it'll be to simplify the code generators that emit assembly, and more or less by definition it only makes sense for it to contain things that are straightforward to cope with.

4.2 Off-the-shelf technology

The following things I think can be handled with off-the-shelf compiler technology, whether RTLs or something else. They are not necessarily trivial or straightforward, but they should at least be tractable.

- **Endianness.** While the endianness itself is trivial, generating the endianness-related functions using native byte-swap instructions is less so. Still, it shouldn't be anything that can't be done with standard compiler technology.

- **Memory barrier operations.** Specifying what the semantics of a memory barrier instruction really are in a way a code generator can reason about isn't exactly trivial (since it's about concurrency and nothing about concurrency is trivial) but in order to generate a library of memory barrier operations with a MI interface we don't need to do that. We only need to do that if we want to try to synthesize our own lock-free data structures and that seems like its own thesis topic.

- **Atomic operations.** This one is a little harder, but for the most part I think the same reasoning applies.

- **`setjmp`.** Everything we need to know to emit `setjmp` and friends follows from the function call ABI. This is true even if the user and kernel versions need to be different because of e.g. floating point registers.

- **Signal frames.** Like `setjmp` this follows from the function call ABI. Unless we want to try to be compatible with an existing implementation (probably a bad idea) it's much the same, though in some sense dual: you need to save all caller-save registers. The trampoline might be a bit less trivial but doesn't seem fundamentally hard. Plus if it comes to it one can do signals without trampolines at the cost of extra system calls.

- **Signal posting code.** This should be a straightforward manipulation of trapframe state based on e.g. knowing what slot the program counter lives in.

- **FPU context save/restore.** This is much like `setjmp` also: dump out some state, reload from somewhere else. Controlling the FPU state to make it work should follow from RTLs. Knowing when to do it based on traps falls under "trap dispatching code" below.

- **Kernel thread switch code.** This is also like `setjmp`.

- **Trapframe updating code for syscall return.**

- **Trapframe updating code for fork return.** These are much the same as signal posting code.

- **Register set declarations for `ptrace`.**

- **Register set declarations for ELF coredumps.** Both of these can just be dumped out given the list of registers, unless we need to be compatible with existing declarations in gdb or similar; in that case we need to have the register order as part of the input machine description.

- **`ptrace` register handling.** This just requires knowing what registers there are.

- **Floating point emulation.** This should be a straightforward code generation problem. In fact, probably we don't have to do anything at all; we should be able to take existing floating point emulation code and feed it through our compiler. At least provided that all we need to care about are IEEE floats.

- **Softfloat code.** Same, just in userland instead of in the kernel and possibly with some ABI issues.

- **MD bits of libm and libc floating-point things.** I would think this would be straightforward given an adequate description of the FPU.

- **Kernel profiling hooks.** I'm not absolutely sure what this entails but I think it's mostly about retrieving the caller of a stack frame, which is standard debugger technology.

- **Userland profiling logic.** Likewise.
- **Spinlocks.** I don't think this is any harder than the atomic operations library; it's just maybe not quite the same.
- **Interrupt on/off code.** This should come directly out of RTL information.
- **Interrupt wait/suspend.** This should come directly out of RTL information; we just need to be sure we can specify the semantics of these instructions correctly.
- **CPU halt.** In the absence of a specific instruction this is just wait in a loop.
- **System call entry stubs.** This is straighforward even if it entails converting from one function call ABI to a different system call ABI; it just needs to know the instruction for issuing a system call and that's part of the system call ABI specification.
- **User startup code.** This is the dual of a system call entry stub, modulo needing to do a few additional things; but those things are well understood.
- **Thread switch code for userlevel threads.**
- **Thread start code for userlevel threads.** These are different from the kernel versions only because they work in a different context; they need to do the same set of things in each case.

4.3 Material given to us.

The following things either pretty much need to be given to us, either directly or as some kind of more concise form specifying what to emit from information already known.

- **Standard or semi-standard MD header files.** Taking the MIPS `regdefs.h` file as an example: the names and register numbers in this come from the ABI definition, which is material we necessarily have; however, the fact that the file needs to exist and the knowledge of what needs to go into it is something we have to be told. We could provide a way to specify additional header files and their contents; or we could just treat the whole file as part of the processor spec. The former is more principled (but somewhat harder) but the latter seems perfectly justifiable.
- **Goop for `float.h` and `fenv.h`.** As far as I know (though I'm a long way from an expert on float and FPU issues) everything in here comes straight from the FPU specification. Which means we need enough of a FPU specification to generate these files; but depending on what we try to do for libm it may be that the right way to do this is just to provide the files, or the information for the files, directly as part of the machine description.
- **ELF relocation codes.** There's no way we can synthesize these as they're externally defined. (In principle we could synthesize a binary format and synthesize relocations based on what the machine code needs; but this is probably neither simple nor interesting unless we want to use randomly generated machines or something like that.)
- **ELF machine type and word-size codes.**
- **Maximum number of cpus.** This pretty much must be a direct part of the platform description.
- **Sizes and usage restrictions of superpages, if any.** This had better just be part of the MMU description.

4.4 Not quite off-the-shelf

- **Exception entry code.** In some cases this will probably require goal-oriented mode search as described above; e.g. on MIPS before you can do anything else you have to locate your kernel stack, which isn't necessarily entirely trivial. After that it's just dumping out registers though which is straightforward.

- **Trap dispatching code.** This requires a list of the trap codes and some specification of their semantics. These can then be mapped to the MI list of things that can happen at trap time (e.g. call the interrupt dispatcher, call the system call dispatcher, do nothing, call the floating point emulator, kill the current process with some signal, etc.) Exactly what the specification of trap semantics should be isn't obvious up front but it seems like it shouldn't be all that difficult.
- **Busywait timing loop.** Generating a loop should be straightforward; timing it perhaps not quite so.
- **Logic for coping with pipeline hazards.** I think it should be sufficient to either stick hazard bits in the RTL state or write a list of constraints on pairs of actions involving particular pieces of state. The latter's probably cleaner.
- **Code for handling cycle counters.** Cycle counters are just more registers. The interesting part is specifying the semantics of what they're counting; some of that can be made to fall out of RTL descriptions, some of it probably less so. However, other than the main counter of cycles elapsed (which we'll want for timing if it exists) much of this may not matter.
- **Kernel thread start code.** This is not quite obvious but I think nearly all of it follows from the function call ABI.
- **`ptrace` single-step handling.** In theory it should be possible to find any single-step mode in the RTL description, and then it's probably straightforward to enable it for userlevel execution in the target trapframe. This seems potentially somewhat fragile though; or at least, once engaged it interacts with the trap dispatching code and how that plays out is not entirely obvious.

4.5 Fairly tractable

- **How to handle `curthread/curcpu`.** There are only a handful of schemes for this; my guess is that after digesting the processor spec we can choose by plotting out how one would do each, rejecting ones that don't work, and scoring the rest by some fairly basic criteria.
- **VM-related constants.** Many of these are basically givens (e.g. `PAGE_SIZE`. Some of the rest (e.g. the bounds on userspace addresses) are either wired into the processor or can be more or less just assumed based on simple criteria. I don't think anything in here is difficult unless we're trying to squeeze every last advantage out, which we aren't.
- **`copyin/copyout`.** Given an MMU description that lets us reason about user addresses vs. kernel addresses, this should be fairly tractable: if there are special instructions for reading from user addresses we should be able to find them, and if there aren't the logic for doing it safely with ordinary read and write instructions is pretty much canned. There are some second-order concerns about doing it efficiently in the latter case; if we don't care about that (true up front, unclear later on) it reverts to an instance of `setjmp`.
- **Cache control logic.** Despite my concerns above I think modeling what we need won't be that hard, and given the modeling, generating cache control instructions is much like anything else. It's the parts of this that are really part of the VM system that concern me more; but that's because I still don't

myself really have a good handle on what a VIPT cache does and doesn't demand from the VM system.

- **System call argument collection.** This is not much different from other things that are derived from function call ABIs; it's a bit more involved becomes sometimes part of the arguments need to be fetched from userspace, and because the form in which the arguments are presented to the MI kernel may need its own specification.

- **Native 32-on-64 compatibility.**

- **Glue for foreign OS system call compatibility.** These just mean extra copies of other things and making sure the other things get slotted into per-system-ABI data structures properly. We don't need to do this; but it's not expensive.

- **MD hooks for big in-kernel things.** Most of these things are instances of debugger or code generator technology and should therefore be mostly off the shelf problems. (If we need them at all.)

- **Any user-level manifestations of cache control logic.** This is a fairly clear subset of the kernel-level cache control problem: at userland you either need nothing, or can issue the same instruction you do in the kernel, or if that instruction is privileged, issue a system call. The catch is knowing what system call to make; we can't just make a private one up, because in the example of `mips_cachectl` the call is known and semi-standard and needs to exist for third-party programs to work. (This doesn't prevent us from making up a private one for our own use and providing the known one only to third-party code, but that's unsatisfying.)

- **Further MD bits of libpthread.** I'm not totally familiar with what's in there but I don't think it's likely to be anything markedly different from what's needed for in-kernel threads. The chief likely problem is dealing with thread-local storage; but I think that ends up being encoded into the compiler, linker, dynamic linker, and function call ABI.

- **MD bits of `libkvm`.** As noted above, this is a small appendix to the VM system.

4.6 Less clear

- **Code for on-chip timers.** Modeling timers seems problematic, so it's not clear how to deal with this.

- **How many interrupt levels to use.** No idea how to synthesize this, but either being told it or always using the same number based on MI properties of the OS seem like viable approaches.

- **Machine-dependent system calls.** There's no way to know that e.g. somebody a long time ago decided it would be a good idea to make the x86 `iopl` instruction available to root user-level processes. We have to be told about these. But it's not at all obvious how to specify them. Probably we can think of them as being like hypercalls in paravirtualization: they have semantics expressible in the same language as the supervisor mode instruction set, plus some signature or calling sequence. It *should* be possible to take lists of declarations of that form and emit the necessary kernel-side instructions. (And many of these we don't actually need to support for the software we need to run.)

- **Kernel startup code.** For x86 this is highly nontrivial, because you have to wire up and enable a lot of processor things. As discussed above this (at a minimum) requires a goal-oriented search through modes with a lot of steps. It interacts with the VM system too, as one of the things that needs to be wired up is pagetables for the kernel. But at least I have some ideas about how one might do it.

- **Compiler and linker flags.** It isn't clear to me how one deduces that compiler and linker flags are needed. Things like `-fno-pic` for mips are probably just givens.

- **Linker script.** Likewise, it's not clear how to deduce what to put into this.

- **ELF dynamic linker.** In theory this is not any worse than the MD parts of a linker, which basically amounts to applying relocations, which is straightforward. However, the dynamic linker is a very adverse program environment and among other things it has its own startup code problem: it can't use anything from libc because it has to be able to run first in order to load libc. I'm not sure exactly what this entails.

- **What to put in the kernel config.** Like many of the other platform-related issues it isn't clear either what needs to be specified or how to specify it. However, regardless of how hard a problem that turns out to be, this particular part of it shouldn't end up being rocket science.

- **Platform-dependent parts of installer.** This I think ends up being mostly the same logic as figuring the kernel config.

4.7 Some traction

- **VM system code and headers.**

  This is pretty much the centerpiece of the project – the part that's nontrivial but hopefully not impossible. It differs from the things for which RTLs will serve in that it involves manipulating a mapping rather than concrete state. (While one could treat a 4MB 32-bit pagetable as 4MB of concrete state, and define RTL for operations on it this isn't very helpful and it doesn't characterize the semantics of the *mapping*, which is what the VM system needs to engage.)

  Note that one can either generate a complete VM system, or just ("just") something akin to a pmap module that plugs into a handwritten MI VM system. Which of these is preferable isn't clear at this stage. Note that this choice has significant implications for the rest of the project: refitting a new VM system to an existing kernel (especially an existing legacy kernel that isn't particularly structured according to my standards) is a big enough undertaking that assembling a new one (even out of roughly the same pieces) is likely a better bargain.

  When I first thought up this project (well over ten years ago) the basic approach I had in mind was to specify a VM system that included support for manipulating all the known different kinds of machine-level mappings, both forward and backward, and then for any given platform optimize out the ones that weren't needed. Part of the point of this was to reduce the redundancy seen in conventional designs between the MI forward (P to V) mapping and the machine-level forward mapping most non-legacy processors specify. And also, in the case of software-fill TLB MMUs like the MIPS, perhaps devise a single forward mapping that could be used efficiently both for fast path and slow path fault processing.

  In the intervening time I've acquired a good bit more experience with VM system design, and in particular the appurtenances required by support for copy-on-write, memory-mapped files, and other major features, and that reduced redundancy no longer seems like a worthwhile goal: while it's to some extent technically possible, it isn't really something one would want; in the presence of (particularly) memory-mapped files it becomes messy and provides only a minimal benefit. So that's no longer in itself an argument in favor of synthesizing the entire VM system.

However, the approach is still valid. Updated a bit based on the fact that in the intervening time I've also acquired a lot more experience with language and compiler issues, it amounts to the following:

First, write a new VM system in a high-level domain-specific language. The language should be high enough level that one can do semantic optimizations, but also still capable of representing memory; this requires some delicate tradeoffs but should be doable. Likely the language shouldn't support heap allocation of language-level objects; in order to make that work it will probably require domain-specific knowledge of allocation contexts and scopes.

Second, write the VM system in terms of manipulating hardware-level forward and backwards mappings. Then, given a way to characterize the representations (or nonexistence) of these mappings on a given machine, we can convert the manipulations of these mappings to assembly code. Then we can compile the rest of the DSL to C or assembler, and that's the result.

How one best handles concurrency in this model is not entirely clear; one could code it directly in the DSL, or one could try to have the DSL compiler synthesize part or all of the locking model. The latter is hard though and should probably be left until the other functionality is under control.

A related approach is to take an existing VM system that is written in terms of "pmap" modules (or equivalent), formalize the existing interface to the pmap layer (this will likely involve quite a bit of cleanup and restructuring) and then proceed in roughly the same fashion.

It isn't obvious to me up front if writing new is better or worse – writing a new VM system is a good chunk of work, but one can also sink huge amounts of time and effort into trying to systematize existing material, especially existing material with unclear ad hoc semantics. (Also, the process of writing a new VM system has guaranteed termination; mucking with an existing one's pmap interface does not. This is also an important consideration.)

A potential bonus of the new VM system and DSL approach is that one could also provide switches to keep or prune and optimize away the big-ticket features like copy-on-write and memory mapped files. This offers trendy advantages in the current atmosphere of cloud deployments and unikernels.

A different approach entirely is to treat it as more of a code synthesis operation: instead of designing in hooks for all anticipated MMUs and then using the machine description to prune unnecessary logic, use some kind of goal-directed search to bulid the higher-level structure based on the lower-level operations and structures available. This kind of thing is hard, though, and constructs the size and complexity of VM systems are I believe several orders of magnitude beyond what anyone's accomplished before in practice. That doesn't mean it can't be done, but there are many unanswered questions and not very many obvious avenues of attack.

4.8 Totally unclear

- **Detecting RAM size.**
- **Properties of different regions of RAM.**
- **How you find secondary CPUs and cores.**
- **`bus_machdep.c` for various buses.**
- **`bus_space`, `bus_dma` and similar material.**

- **Interrupt identification and dispatching code.**
- **Bus-level interrupt masking code.** All of these are platform-dependent and depend on bus- and system-board-level material that I don't know how to specify and haven't really thought much about yet.
- **Early console.**
- **Early disk I/O.**
- **Interface to firmware.**
- **Bootloader install tool.**
- **Kernel image conversion tools.** These are even worse: they depend on firmware issues.
- **CPU identification logic.**
- **MD bits of `cpuctl`.** These two are mostly the same problem, but I'm not at all sure how to deal with it.
- **Workarounds for CPU bugs.** This one I haven't a clue about.
- **JVM.** Ideally this one would be someone else's problem. Unfortunately, for what we're doing we really need to have it and have it working. This worries me, because JVMs are not exactly known for being clean or portable, and retargeting a JIT is not trivial. Furthermore, using LLVM's JIT is probably not an option; as of the last I heard (fall 2015) LLVM's JIT was not really suitable for this kind of use.

4.9 Hopefully someone else's problem.

- **Bootloader.** x86 bootloaders in particular are nontrivial and have horribly tight constraints. Synthesizing one would be a tour de force; if I can figure out how to do it by the late stages of the project I'm happy to give it a shot, but I'm not optimistic about it being possible.
- **Binutils.** We are in a position to be able to provide a linker, assembler, and disassembler with most likely a reasonable amount of work and time investment. The rest of binutils is trivial.

  At the moment doing this also looks like a good initial exercise in sorting out the preexisting tools and techniques; those are theoretically capable of handling such tools already, although some gaps have already surfaced. (These gaps appear in the space between "an arbitrary assembler for platform X" and "a drop-in replacement assembler that understands the output of existing compilers for platform X and can be used in production".)

  There are also non-research reasons I would like to do these tools; specifically, right now GNU binutils is the only available option ready for prime time and it is horrible, horrible legacy code.
- **Compiler.**
- **Debugger.** We are not, however, in a position to easily provide either a compiler or debugger, and these items are pretty much fundamentally off topic. I think the right thing to do is to declare these pieces out of scope, note that retargetability of compilers and debuggers is fairly well understood in practice, and use an existing retargetable compiler and debugger that's reasonably well engineered. Clang (based on LLVM) fits this description for the compiler; the LLVM debugger (lldb) is probably also suitable. My understanding is that the assembler and linker set for LLVM is not really ready for prime time yet.
- **Lint or other program analyzers.** Providing the necessary MD pieces for BSD lint is trivial. Not sure about sparse and splint; but if they become problematic we can skip them. Note that if we use LLVM we get both

clang-static-analyzer and Klee for (nearly) free, which is an attractive proposition.

4.10 Something completely different

- **Drivers for any platform-specific hardware.** Synthesizing drivers is a different problem from anything else here (except maybe the VM system, as the VM system is sort of a driver for an idiosyncratic hardware device) and it seems to me that it's out of scope; or at least, it's an additional, nontrivial, project. Judging by the speed at which NICTA's driver synthesis project has been going it is not something we should take up if I want to graduate in the next ten years, especially since (unlike the VM system) I don't have any particular ideas about how to do it. While I have pointed this out several times, unfortunately it seems that the message did not go through and we may be stuck with it.