Principia Softwarica: The ARM Emulator `5i`

version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
???

April 28, 2026

Contents

1 Introduction **6**
- 1.1 Motivations . . . 6
- 1.2 The Plan 9 ARM emulator: `5i` . . . 7
- 1.3 Other emulators . . . 8
- 1.4 Getting started . . . 9
- 1.5 Requirements . . . 10
- 1.6 About this document . . . 10
- 1.7 Copyright . . . 10
- 1.8 Acknowledgments . . . 11

2 Overview **12**
- 2.1 Computer principles . . . 12
  - 2.1.1 Von Newmann architecture . . . 12
  - 2.1.2 Registers . . . 13
  - 2.1.3 Instruction format . . . 13
  - 2.1.4 Hardware versus software . . . 14
  - 2.1.5 IO . . . 14
  - 2.1.6 Interrupts . . . 15
  - 2.1.7 Virtual memory . . . 15
  - 2.1.8 Endianess . . . 15
  - 2.1.9 Two's complement . . . 15
  - 2.1.10 Condition codes . . . 16
  - 2.1.11 Why study a 1985 architecture in 2024? . . . 16
- 2.2 Emulator principles . . . 17
  - 2.2.1 Emulator vs simulator . . . 17
  - 2.2.2 User-mode vs full-system emulation . . . 17
  - 2.2.3 Interpretation styles . . . 18
  - 2.2.4 Memory model . . . 19
- 2.3 `5i` interfaces . . . 19
  - 2.3.1 Command-line interface . . . 20
  - 2.3.2 Debugger interface . . . 20
- 2.4 `helloworld` . . . 20
- 2.5 Input binary executable . . . 21
- 2.6 The ARM architecture . . . 21
- 2.7 Code organization . . . 22
- 2.8 Software architecture . . . 23
- 2.9 Book structure . . . 23

3 Core Data Structures **25**
3.1 `Instruction` and `Opcode` . . . 25
3.2 `Inst` and `itab` . . . 26
3.3 `Registers` and `reg` . . . 27
3.4 `Segment` and `memory` . . . 28

4 `main()` **31**
4.1 `main()` skeleton and globals . . . 31
4.2 `inithdr()` . . . 32
4.3 `initmemory()` . . . 33
4.4 `initstk()` . . . 35
4.5 `cmd()` . . . 37

5 Instruction Interpreter **39**
5.1 Instruction binary format . . . 39
5.2 `run()` . . . 40
5.3 `arm_class()` . . . 41
5.4 Arithmetic and logic . . . 42
5.4.1 Opcode extraction . . . 42
5.4.2 Multiplication . . . 44
5.4.3 `Idp()` and `dpex()` . . . 45
5.4.4 Boolean logic . . . 46
5.4.5 `S` bit . . . 47
5.4.6 Addition and overflow . . . 47
5.4.7 `shift()` . . . 48
5.4.8 Immediate values . . . 52
5.4.9 64 bits target (signed/unsigned) multiplication . . . 53
5.4.10 Misc instructions . . . 54
5.5 Memory . . . 55
5.5.1 Opcode extraction . . . 55
5.5.2 Memory swap . . . 56
5.5.3 Load/store . . . 56
5.5.4 Multi registers load/store . . . 60
5.6 Control flow . . . 61
5.6.1 Opcode extraction . . . 62
5.6.2 Simple branching . . . 62
5.6.3 Branch and link . . . 62
5.6.4 Comparisons . . . 63
5.6.5 ARM Conditional execution . . . 63
5.7 Software interrupt . . . 66
5.8 Unimplemented instructions . . . 66

6 Memory **67**
6.1 `page_of_vaddr()` . . . 67
6.2 Page faults . . . 68
6.3 `Tlb` . . . 69
6.4 `ifetch()` . . . 70
6.5 The instruction cache . . . 71
6.6 `getmem_xxx()` . . . 71
6.7 `putmem_xxx()` . . . 73

7 Syscalls Emulation **75**
7.1 `memio()` . . . 77
7.2 Nop and errstr syscalls . . . 78
7.3 Memory syscalls . . . 79
7.4 File and IO syscalls . . . 80
7.5 Directory syscalls . . . 83
7.6 Namespace syscalls . . . 85
7.7 Misc syscalls . . . 86
7.8 Unsupported syscalls . . . 88

8 Debugger **91**
8.1 Overview . . . 91
8.2 Interface . . . 93
8.2.1 Inspecting: `$` . . . 95
8.2.2 Controlling: `:` . . . 97
8.3 Format . . . 98
8.4 Dumpers . . . 103
8.5 Traces . . . 103
8.5.1 Syscalls trace . . . 103
8.5.2 Stack trace . . . 104
8.5.3 Call tree trace . . . 106
8.6 Breakpoints . . . 107
8.6.1 Code breakpoint . . . 110
8.6.2 Memory breakpoint . . . 110

9 Profiler **111**

10 Advanced Topics **116**
10.1 System instructions . . . 116
10.2 Coprocessor instructions . . . 116
10.3 Float instructions . . . 116
10.4 Signals . . . 116
10.5 Optimisations . . . 117

11 Conclusion **118**
11.1 Patterns and techniques . . . 118
11.2 Connections to other books . . . 118
11.3 Beyond the Plan 9 emulator . . . 119

A Debugging **120**

B Error Management **122**

C Utilities **123**
C.1 Memory Management . . . 123

D Extra Code **124**
D.1 `machine/5i/` . . . 124
D.1.1 `machine/5i/arm.h` . . . 124
D.1.2 `machine/5i/icache.c` . . . 126
D.1.3 `machine/5i/globals.c` . . . 126

D.1.4 machine/5i/utils.c . . . 127
D.1.5 machine/5i/bpt.c . . . 127
D.1.6 machine/5i/mem.c . . . 128
D.1.7 machine/5i/symbols.c . . . 128
D.1.8 machine/5i/run.c . . . 128
D.1.9 machine/5i/5i.c . . . 130
D.1.10 machine/5i/stats.c . . . 130
D.1.11 machine/5i/cmd.c . . . 131
D.1.12 machine/5i/syscall.c . . . 131

Glossary **134**

Index **135**

References **141**

Chapter 1

Introduction

The goal of this book is to present with full details the source code of a processor emulator.

This book plays an unusual role in the Principia Softwarica series. The other books describe software—assemblers, linkers, shells, compilers, kernels—that ultimately runs on a processor. This book describes the processor itself, but expressed as a piece of software (an emulator) rather than as a hardware description. The payoff is that the same literate-programming machinery used to explain `5a` and `5l` in the Assembler book [Pad15a] and Linker book [Pad15b] can be applied to the ARM ISA they target: you can read a C function and learn exactly what a given instruction does to the machine's registers, flags, and memory. In effect, the book is an executable ARM reference manual, and the accompanying Assembler book [Pad15a] and Linker book [Pad15b] books become much easier to follow with it open on another screen.

1.1 Motivations

Why a processor emulator? Because I think you are a better programmer if you fully understand how things work under the hood, and the processor is really at the bottom of what is under the hood.

Every other books in Principia Softwarica will describe programs that ultimately runs on a concrete machine. For the assembler and linker, I will even describe programs that generate binary codes for a specific architecture (the ARM). It is thus useful, especially to understand the assembler, linker, and also compiler, to have somewhere the full description of the instruction set of an architecture (ISA): its binary format, mnemonic names, and also semantics. In fact, the hardware can also be seen as a kind of software: a processor is an interpreter for a low-level language, and so it is also *softwarica* material. The binary format of some instructions and their semantics can be described simply by explaining the code of an interpreter: a processor emulator.

Even if most programmers never write an emulator, understanding how a processor works at this level has immediate practical value. When debugging a crash, the backtrace makes more sense if you understand how the stack frame is laid out in memory and how the link register saves the return address. When optimizing hot loops, knowing which operations the hardware can do in a single instruction—and which require multi-step sequences—guides better decisions than any profiler alone. And when reading the other books in Principia Softwarica, this book serves as a companion to Assembler book [Pad15a] and Linker book [Pad15b]—providing the ISA reference that those books assume—expressed as a literate program rather than as a 2000-page architecture reference manual.

Here are a few questions I hope this book will answer:

- What is the list of all processor instructions? What can a typical computer do?
- What are the most important instructions? How can they be used to implement higher level constructs?
- Which instruction allows to enter in kernel mode?
- How instructions are encoded? What is the binary format?

- How is handled overflow? What are the differences between the logic shift-right and arithmetic shift-right operations?
- How does an ARM machine compare to a Turing machine? To a Von Newman machine?

1.2 The Plan 9 ARM emulator: `5i`

I will not describe in this book a processor in the form of a program of an hardware description language (e.g., VHDL), which operates at the granularity of logic gates (e.g., `or`, `and`, `nand`). For such an approach I recommend the great book *The Elements of Computing Systems* [NS05] and its companion website `http://www.nand2tetris.org`. Instead, I will present the source code of an *emulator* written in the high-level language C, which allows us to describe a relatively complex processor and its machine language in a book of a reasonable size. An emulator can be viewed as an executable specification[1] of a machine.

I will explain in this book the code of the ARM emulator `5i`[2] which is about 4400 lines of code (LOC). `5i` emulates the execution of an ARM binary in a Plan 9 environment. The `5` comes from the Plan 9 convention to name architecture with a number or single letter (0 is MIPS, 5 is ARM, 8 is x86, etc), and the `i` probably means interpreter.

I chose the ARM architecture over the x86 architecture because even if x86 is the processor of most desktop machines today, ARM has a far simpler architecture. Indeed, RISC (Reduced Instruction Set Computer) machines such as an ARM, as their name suggest, have a smaller set of instructions than CISC (Complex Instruction Set Computer) machines such as an x86, and have also simpler instructions. The code of an ARM emulator is thus smaller and simpler to describe, while still conveying in my opinion the essence of all processors.

I chose ARM over MIPS, because even if the MIPS is a RISC machine probably simpler than the ARM, there is not much remaining MIPS machines around. The ARM on the other hand is very much alive; it is the most popular processor in phones today. It is also the processor of the extremely cheap Raspberry Pi[3] machine, a machine used by many electronic hobbyist. This also makes ARM a great candidate for our teaching purpose.

Note that `5i` is not a complete processor simulator though. It emulates all the basic instructions, and so is a good reference for the ARM ISA, but it does not emulate low-level things such as hardware interrupts, device interactions (IO), coprocessor instructions, kernel and user modes, or the booting process. The only system instruction partially handled is the software interrupt, and it is emulated by mimicing the semantics of the Plan 9 system calls[4]. `5i` can run simple Plan 9 programs compiled for the ARM, but it can not run a kernel. However, I think `5i` is a good compromise for our educational purpose.

In other words, `5i` is a user-level emulator: it simulates the CPU as seen by an application, not as seen by the kernel. This is the same approach used by tools like QEMU's user-mode and, in a different context, Wine—intercept system calls at the boundary and let the host OS handle them, rather than simulating the entire hardware platform.

Note also that describing a C program, and not a VHDL program, has a few disadvantages. Indeed, to emulate for instance the `AND` ARM instruction, I will show C code such as:

⟨*instruction interpreter simplified snippet 7*⟩≡
```
case AND_INSTRUCTION:
  register3 = register1 & register2;
  ...
  break
```

[1] This program will be useful also for testing purpose when I will describe in other books the assembler, linker, and compiler.

[2] `http://plan9.bell-labs.com/magic/man2html/1/vi`, which despite its name covers also `5i`

[3] `https://www.raspberrypi.org/`

[4] As a side effect, this book also helps to understand the kernel, especially its API and the semantic of a few important system calls such as `sysbrk()`.

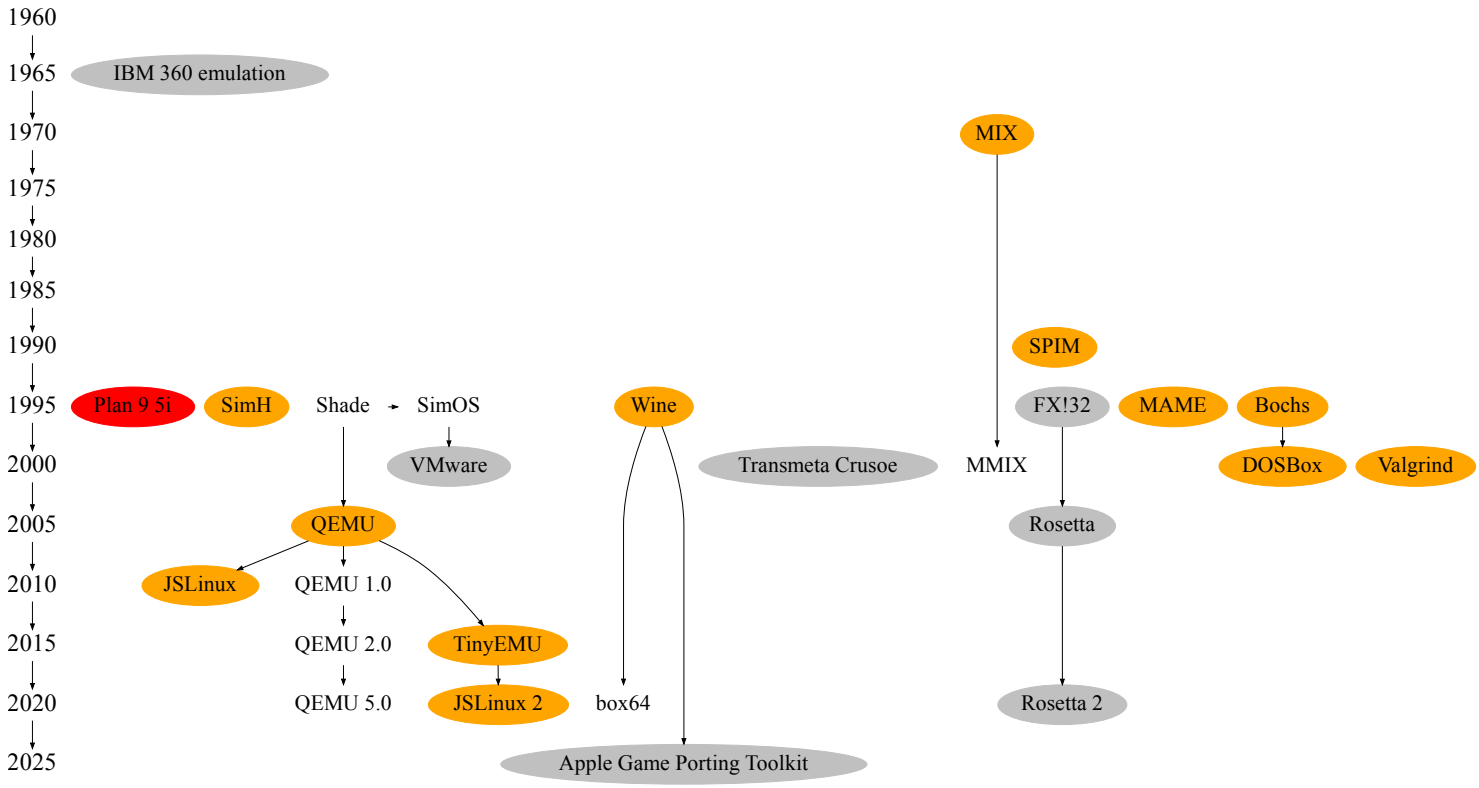


Figure 1.1: Emulators and instruction set simulators timeline

This code uses the C binary `&` operator. I am cheating in some sense, just like when someone describes the code of a Lisp interpreter written in Lisp; it does not help to understand fully how Lisp is implemented, for instance, to understand how a garbage collector works. In our case, using C does not help to understand how certain operations work at the bit level, how for instance a 32 bits adder, or multiplier, or multiplexer works[5]. Is is still useful though to get an overview of the fundamental capabilities and features offered by most computers and to better understand the ARM ISA.

1.3 Other emulators

Here are a few emulators that I considered for this book, but which I ultimately discarded:

- SPIM[6] is a MIPS processor emulator. It is one of the first emulator written for educational purposes. It is partially described in an appendix of the classic architecture book *Computer Organization and Design: The Hardware/Software Interface* [PH13]. However, as I mentioned it earlier, MIPS is unfortunately a dying machine. Moreover, SPIM does not execute *binary* MIPS programs; instead, it interprets *assembly* MIPS programs. This hybrid approach would then force me to describe the SPIM assembly scanner and parser, instead of focusing on the binary format of instructions, which is in my opinion an important thing to understand for Principia Softwarica. Finally, the 15 000 LOC of SPIM are bigger than the 4200 LOC of `5i`.

- QEMU[7] is a popular and fast emulator supporting many architectures, including ARM and x86. Instead of the simple interpretation approach used by `5i`, QEMU uses dynamic binary translation, which improves

[5] Again I recommend [NS05] for that.
[6] http://pages.cs.wisc.edu/~larus/spim.html
[7] http://www.qemu.org

a lot its performance. QEMU emulates everything (system instructions, faults, devices, etc), and so can run entire kernels. In fact, I suggest you to use QEMU to experiment with Plan 9. However, the codebase of QEMU is very large: more than 1.2 million LOC in total. Even its `hw/arm` subdirectory is already more than 25 000 LOC.

- MAME[8] is a popular emulator for arcade machines (e.g., R-type, Pong), consoles (e.g., Atari 2600, Nes, Gameboy, NeoGeo), calculators (e.g., ti85), and vintage computers (e.g., Alto, z80, Atari ST, Amiga). It is used mainly to play old video games (programmed for old machines) on modern computers. It supports hundreds of such machines and can run thousands of video games. But this generality has a price; its codebase is very big, almost 5 millions LOC. It would be too hard to describe this emulator in a book, even if I would focus just on one specific machine.

- Hack[9] is a very simple machine described in [NS05] and used for educational purpose. Its emulator consists of 7000 LOC. It also comes with a very nice debugger of 6000 LOC. Hack is a great resource to learn about architecture and to understand how a simple processor works. However, the processor is too restricted and arguably too simple for Principia Softwarica. Moreover, the Hack machine does not exist for real, and programmers have written very few tools for it. For instance, there is neither a C compiler targeting this architecture nor a real operating system for it.

- MMIX[10] and its ancestor MIX are computers designed by Donald Knuth and used in his classic book series *The Art of Computer Programming* [**?**]. Donald Knuth also wrote A book using literate programming explaining the full code of an MMIX emulator [Knu99] (including a description of the processor pipeline, the floating point unit, tje assembler). Howoever, similar to Hack, there are very programs for this machine. For instance, Donald Knuth in his books assumes the presence of an operating system called NNIX, but nobody has ever written it.

- TinyEMU[11] is a RISC-V and x86 system emulator also written by Fabrice Bellard (the author of QEMU). Unlike QEMU, TinyEMU was designed from the start to be small and simple while remaining complete enough to boot Linux. It targets RISC-V, a modern open-source RISC architecture that is gaining momentum as a successor to ARM in the embedded and academic worlds. Plan 9 has even been ported to RISC-V, so TinyEMU would have been a tempting alternative to `5i` for this book. TinyEMU is also the engine behind JSLinux[12], which runs Linux in a web browser by compiling the emulator to WebAssembly. However, at around 40 000 LOC, TinyEMU is still an order of magnitude larger than `5i`'s 3000 LOC.

Figure 1.1 presents a timeline of major emulators and instruction set simulators. I think `5i` represents the best compromise for this book: it implements the core of an ARM emulator—instruction decoding, register operations, memory access, and system call proxying—while remaining small enough (about 3000 LOC) to explain in full.

1.4 Getting started

To play with `5i`, you will first need to install the Plan 9 fork used in Principia Softwarica. See https://www.principia-softwarica.org/. `5i` is also available through Goken9cc[13], where it can be compiled natively on Linux, macOS, or Windows using gcc or clang. Once installed, you can test `5i` under Plan 9 with:

[8] http://www.mame.net/
[9] http://www.nand2tetris.org/05.php
[10] http://www-cs-faculty.stanford.edu/~uno/mmix.html
[11] https://bellard.org/tinyemu/
[12] https://bellard.org/jslinux/
[13] https://github.com/aryx/goken9cc

```
[1] $ cd /tests/5a/
[2] $ 5a helloa.s
[3] $ 5l helloa.5 -o helloa
[4] $ 5i helloa
[5] 5i> :c
[6] hello world
[7] exists(end)
[8] stopped at #1068 _main+48
[9] $
```

The commands in lines 2 and 3 respectively assembles and links the very simple `helloa.s` hello-world ARM assembly program (or cross-assemble and cross-link if you are on a non-ARM host machine). Line 4 runs the emulator on the ARM binary program `helloa`. `5i` has an interface similar to a debugger, and so the input command `:c` entered after the `5i>` prompt is used to tell the emulator to continue the execution of the emulated program `helloa`. `5i` then interprets the program, which outputs `hello world` at line 6. `5i` then outputs a few debugging information at lines 7 and 8 before exiting.

1.5 Requirements

Because most of this book is made of C source code, you will need a good knowledge of the C programming language [KR88] to understand it.

Note also that this book is not an introduction to computer architecture. I assume you already have a basic understanding of how a processor works, and so that you are familiar with concepts such as registers, memory addressing modes, interrupts, etc. I assume you already know most of the theory; this book is here to cover the practice. See [Tan88, PH13] instead for excellent introductions to computer architecture.

It is not necessary to know the ARM architecture to understand this book. In fact, this book can be used as an introduction to the ARM processor. However, I highly recommend to print the excellent colorful ARM reference card `http://re-eject.gbadev.org/files/armref.pdf`. It will help you to visually understand the binary format of the ARM instructions. This card is especially helpful to understand the code that does many bit manipulations to generate the different parts an ARM instruction.

If, while reading this book, you have specific questions on the command-line interface of `5i` or on the related Plan 9 debuggers, I suggest you to consult the manual pages in my Plan 9 repository: `docs/man/1/vi` for `5i` itself, and `docs/man/1/db` for the `db` debugger whose command grammar inspired the `5i` REPL. The ASSEMBLER book [Pad15a] is the natural companion to this book; `assemblers/docs/asm.pdf` ("A Manual for the Plan 9 Assemblers") in my Plan 9 repository is also a useful cross-reference for the instruction encodings emulated by `5i`.

1.6 About this document

This document is a *literate program* [Knu92]. It derives from a set of files processed by a tool, `syncweb` [Pad09], generating either this book or the actual source code of the program. The code and its documentation are thus strongly connected.

1.7 Copyright

1.8 Acknowledgments

I would like to acknowledge of course the author of `5i`, who wrote in some sense most of this book.

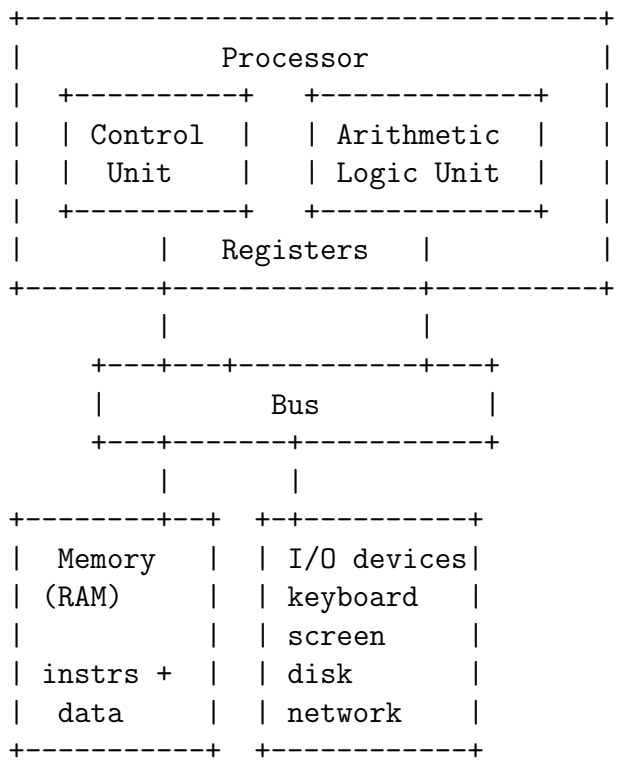
Chapter 2

Overview

Before showing the source code of `5i` in the following chapters, I first give an overview in this chapter of the general principles of a computer and its processor, of the ARM architecture emulated by `5i`, and of the structure of the emulator itself. I also define terms, explain how the code is organized, and more generally give the background necessary to understand the code I will show later.

2.1 Computer principles

A computer in its most general sense consists of a processor, memory, and input/output devices (keyboard, screen, disk, network). The processor executes instructions; memory stores both the instructions and the data they operate on; and I/O devices connect the machine to the outside world.



2.1.1 Von Newmann architecture

The key idea of the Von Neumann architecture is the stored-program concept: instructions and data live in the same memory. The processor reads an instruction from memory, executes it, and moves to the next one.

This is essentially what a universal Turing machine does—it reads a program from its tape and interprets it—except that a real processor provides a richer instruction set than a Turing machine's simple read/write/move operations.

The architecture is named after John von Neumann, who described it in his 1945 draft report on the EDVAC—though Eckert and Mauchly had many of the ideas first, and the attribution has been debated ever since. The main alternative is the *Harvard architecture*, which uses separate memories (and separate buses) for instructions and data; it was common in early computers and survives today in DSP processors and microcontrollers (the AVR chip inside the Arduino is Harvard). Most modern general-purpose processors (x86, ARM, RISC-V) are Von Neumann at the programming-model level but have separate instruction and data *caches*, a compromise sometimes called *modified Harvard*.

What goes into this instruction set? Most processors provide four broad categories of instructions: arithmetic and logic (add, subtract, multiply, AND, OR, shift), memory access (load a value from memory into a register, store a register back to memory), control flow (branch to a different instruction, conditionally or unconditionally), and system instructions (trigger a software interrupt to enter the kernel). Some processors also include floating-point instructions, though historically these were handled by a separate coprocessor. On ARM, memory access follows a load/store discipline: arithmetic instructions operate only on registers, never directly on memory.

2.1.2 Registers

Memory is slow. Even with a cache, reading from main memory takes tens or hundreds of CPU cycles—far too much for the inner loop of a program that reads and writes its working variables on every instruction. Every processor therefore provides a small, dedicated bank of registers: fast storage locations inside the CPU itself, named directly by a few bits inside each instruction. On a load/store architecture like ARM, arithmetic instructions can only operate on registers, so the compiler spends a good deal of effort on register allocation: deciding which variables to keep in registers and which to spill to memory.

The number of registers a processor exposes is a design trade-off. More registers means less spilling and faster code, but also more bits per instruction to encode the register number, which either enlarges the instruction or leaves fewer bits for everything else. x86 (1978) had eight 32-bit general-purpose registers named `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP`, `ESP`; x86-64 doubled that to sixteen by adding `R8` through `R15`. Classic MIPS (1985) and RISC-V (2010) both went with thirty-two registers, using five bits per register field. ARM (1985) sits in between with sixteen—a four-bit field per register—precisely because those extra four bits were spent on the condition code and the `S` bit that the next subsections will introduce.

ARM's register file is unusual in three other ways. By convention `R13` is used as the stack pointer and `R14` as the link register holding the return address for a `BL` (branch and link) call. These are not hardware-reserved—a program is free to use them as general-purpose registers—but every piece of system software assumes the convention. More surprisingly, `R15` is the program counter, directly exposed to user instructions. Writing to `R15` is a jump; reading it returns the address of the currently executing instruction plus eight, because of the original ARM1 prefetch pipeline. So `ADD R15, R15, #4` is literally an unconditional branch, and many of ARM's addressing modes use `R15` as an implicit base register to get PC-relative data access. x86, MIPS, and RISC-V all hide the PC from ordinary instructions, which is the more common design.

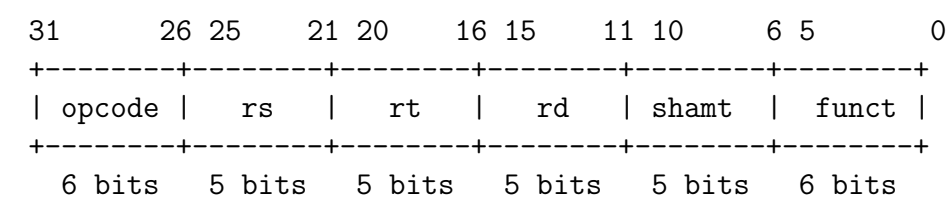
Section 3.3 shows how `5i` represents the register file: a `Registers` struct with 16 `u32int` slots plus a `CPSR` for the flags. Every instruction handler in the interpreter chapter reads and writes this struct directly, and has to special-case `reg[15]` in exactly the ways ARM's architecture reference manual prescribes—which is where the "PC plus eight" quirk starts to matter.

2.1.3 Instruction format

Every instruction must be encoded as a sequence of bits—the binary format that the processor reads from memory. On the ARM, all instructions are exactly 32 bits (4 bytes) wide, which greatly simplifies decoding compared to variable-length architectures like x86. Within those 32 bits, different bit fields encode the opcode

(which operation to perform), the operands (which registers or immediate values to use), and on ARM, a condition code that makes every instruction conditionally executable.

Here is the layout of a MIPS R-type instruction, which illustrates the general idea:



The `opcode` selects the instruction family, `rs` and `rt` are the two source registers, `rd` is the destination register, and `funct` refines the operation (e.g., add vs. subtract). ARM uses a similar fixed-width scheme but devotes the top four bits to a condition code, which makes every instruction conditionally executable—an unusual feature among RISC architectures.

The fixed-width-vs-variable-length choice tracks the RISC *vs* CISC divide that shaped processor design in the 1980s. CISC architectures (x86, VAX, Motorola 68000) pack many complex addressing modes into variable-length instructions, making the decoder large but saving code space. The RISC *revolution*—Berkeley's RISC-I (1982), Stanford's MIPS (1981), and Acorn's ARM (1985)—showed that simpler, fixed-width instructions execute faster in a pipeline, and that the *simplicity* gained in hardware more than compensates for the extra instructions the compiler must emit. Today the distinction has blurred: x86 survived on the strength of its enormous installed software base despite its CISC heritage, and internally decodes complex instructions into RISC-like micro-ops; ARM added a variable-length mode (Thumb/Thumb-2) for code density. But the pendulum is swinging back: Apple's M1 chip (2020) showed that a genuine ARM RISC design could match or beat x86 on desktop performance-per-watt, and server ARM (AWS Graviton, Ampere) is taking datacenter share from x86 for the first time. The RISC idea won twice: first *inside* x86 (as micro-ops), then by threatening to *displace* x86 altogether.

2.1.4 Hardware versus software

Why are certain operations implemented in hardware rather than in software? The obvious reason is speed: 32-bit addition in a single clock cycle is vastly faster than simulating it with a loop of bit operations. But there is a subtler criterion: an operation belongs in hardware when it is used so frequently and so universally that the silicon cost is justified. Arithmetic and branching are the most basic examples. Floating-point was long considered optional (it was a separate coprocessor chip on early machines), but became standard as scientific and graphics workloads grew. Some instructions exist specifically to support the compiler (stack push/pop) or the kernel (save/restore all registers on context switch).

The boundary has shifted with each hardware generation. The floating-point coprocessor (a separate chip on early x86 and ARM boards) was integrated onto the main die in the 1990s; SIMD vector extensions (Intel's SSE in 1999, ARM's NEON in 2004) moved parallel-data operations into hardware; and GPU compute (NVIDIA CUDA 2007, OPENCL 2009) pushed massively parallel workloads onto a dedicated processor. `5i` emulates only the integer ARM core—no floating-point, no NEON, no GPU—which is why the emulator stays small and the principles above are all it needs.

2.1.5 IO

A processor that can only compute on memory is useless without a way to interact with the outside world. Input/Output (I/O) is what makes a computer more than a calculator: it connects the processor to keyboards, screens, disks, and networks. There are two main approaches to I/O: memory-mapped I/O, where device registers appear at special memory addresses (the approach used by ARM and Plan 9), and dedicated I/O instructions (the approach used by x86 with its `in`/`out` instructions). Note that `5i` does not emulate I/O devices—it handles only the software interrupt that triggers system calls.

2.1.6 Interrupts

Interrupts allow external devices (a keyboard press, a timer tick, a disk completion) to notify the processor asynchronously, without the processor having to continuously poll each device. When an interrupt occurs, the processor suspends the current program, saves its state, and jumps to a handler address found in an interrupt vector table. Software can also trigger interrupts explicitly via a software interrupt instruction (called `SWI` on ARM), which is the mechanism used to make system calls—and the only "system" instruction that `5i` actually emulates.

2.1.7 Virtual memory

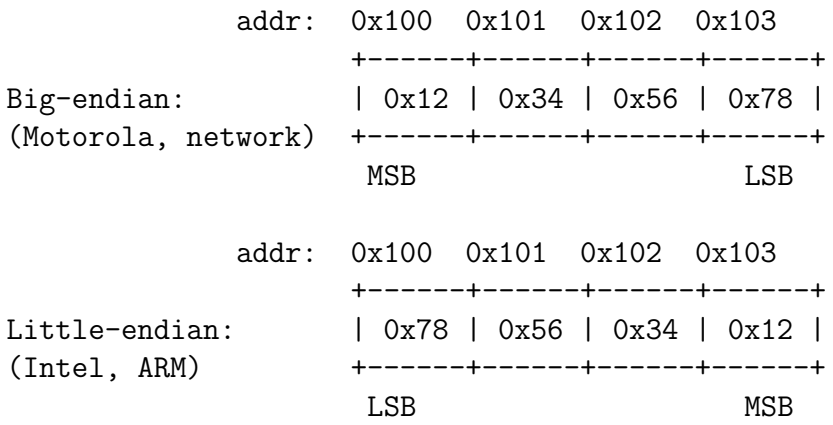
Virtual memory is a hardware mechanism that gives each process the illusion of having its own private address space. The minimal hardware support required is a TLB (Translation Lookaside Buffer) that maps virtual addresses to physical addresses, and a page fault interrupt when a mapping is missing. The kernel handles the fault by updating the page tables and resuming the program. Virtual memory is closely tied to protection: the hardware enforces a distinction between kernel mode and user mode, preventing user programs from accessing each other's memory or the kernel's data structures. None of this is emulated by `5i`, which runs entirely in user space on the host—see KERNEL book [Pad14] for the full story.

2.1.8 Endianess

Endianness determines the order in which the bytes of a multi-byte value are stored in memory. In a little-endian machine, the least significant byte is stored at the lowest address; in a big-endian machine, the most significant byte comes first. ARM is technically bi-endian—it can operate in either mode—but in practice (and in Plan 9) it runs little-endian.

Big-endian is actually the more natural format for humans: the number `0x12345678` is stored as `12 34 56 78` in memory, in the same left-to-right order we write it. Network protocols (TCP/IP) adopted big-endian early on—hence the term "network byte order"—and so did Motorola (68000) and IBM (System/360). Little-endian, on the other hand, is more convenient for hardware: adding one byte to a value at a given address does not change the address of the least significant byte, which simplifies variable-width arithmetic. Intel chose little-endian for the 8080 and x86, and ARM followed. The split persists because neither format is strictly superior, and once an architecture picks one, backward compatibility locks it in.

For the 32-bit value `0x12345678` stored at address `0x100`:



2.1.9 Two's complement

Like all modern processors, ARM uses two's complement to represent signed integers: a negative number $-n$ is stored as $2^{32}-n$, so -1 is `0xFFFFFFFF` and -10 is `0xFFFFFFF6`. The beauty of this encoding is that the hardware `ADD` instruction works identically for signed and unsigned operands—the bit pattern of the result is the same

either way. The difference only matters when interpreting overflow: adding two large unsigned numbers may carry past 2^{32}, while adding two positive signed numbers may wrap around to a negative result. ARM tracks both cases via separate flags (`C` for carry, `V` for signed overflow), but only updates them when the `S` bit is set (see Chapter 5).

2.1.10 Condition codes

Most arithmetic instructions do not just produce a result; they also update a small set of single-bit condition codes (also called flags or status bits) that summarize properties of the result. Conditional branches and, on some ISAs, conditional execution of arbitrary instructions, then read those bits to decide what to do next. The classic four flags, shared by almost every ISA that has them, are:

```
N   result was negative       (= high bit of result)
Z   result was zero
C   unsigned carry or borrow out of the high bit
V   signed overflow (two's complement wraparound)
```

ARM stores them in the top four bits of the current program status register (`CPSR`); x86 has them in its flags register (`EFLAGS` or `RFLAGS`); PowerPC has eight four-bit condition register fields (`CR0`–`CR7`).

The reason this tiny bundle of bits gets a dedicated register is that comparisons and conditional jumps use it as their sole communication channel. Writing `if (a < b) goto L` in the source language becomes, in machine code, subtract `a` from `b` followed by branch if the flags say the result was negative. The subtraction is the comparison; the flags are its output. With N and Z alone you can encode equal/not-equal/less-than/ greater-than for signed integers; adding C and V lets you do the same for unsigned values and distinguish signed from unsigned overflow. Almost every ISA with flags implements the same set of 14-to-16 named condition codes (`EQ`, `NE`, `LT`, `GE`, `HI`, `LS`...) that are just boolean expressions over these four bits.

Not every architecture takes this route. `RISC-V` deliberately has no flags register at all: its designers argued that flags create a serialization hazard on superscalar processors (every arithmetic instruction writes a single shared register that later branches read) and instead provide compare-and-branch instructions like `BLT rs1, rs2, label` that do the comparison and the branch in one step. MIPS made the same choice decades earlier. The cost is that multi-word arithmetic (add with carry) becomes harder to express, but the benefit is a cleaner out-of-order execution model.

ARM pushes the flag design to its extreme. Most instructions do not update flags by default—only the ones whose encoding sets the `S` bit do. This lets the compiler emit arithmetic whose flags survive for several instructions, so that a later `BEQ` can still see the result of an earlier `CMP`. ARM also takes the flag idea further than any other mainstream ISA by offering conditional execution of ordinary instructions: almost every ARM instruction has a 4-bit condition-code field, so `ADDGT` means "add if the flags say greater-than", turning a branchless `if` into two instructions. Section 5.6.5 shows how `5i` handles this by checking the flags before every instruction regardless of whether it happens to be a real conditional.

2.1.11 Why study a 1985 architecture in 2024?

The reader may wonder why a book written today teaches a processor architecture from 1985 through a software emulator. The answer is that the *programming model* has not changed. Registers, load/store memory access, condition flags, branches, a linear address space, and the fetch-decode-execute loop that ties them together are exactly the same abstractions that an x86-64, an Apple M4, or a RISC-V chip presents to software today. What modern processors add—out-of-order execution, speculative branch prediction, multi-level cache hierarchies, simultaneous multithreading—is deliberately *invisible* to the programmer; the hardware goes to extraordinary lengths to maintain the illusion that instructions execute one at a time in order, because that is the model compilers and operating systems are written against. `5i` strips away the microarchitectural complexity and

shows *just the model*. Understanding it here, on a machine simple enough to read end to end, is understanding the abstraction that every modern CPU still presents to your code.

2.2 Emulator principles

An emulator is essentially an interpreter for machine code: it reads an instruction, decodes it, executes the corresponding operation on simulated state (registers, memory), and advances to the next instruction. This is exactly the fetch-decode-execute loop that a real processor performs in hardware—except that `5i` does it in a C `while` loop.

2.2.1 Emulator vs simulator

The word "emulator" is used loosely in practice, but in the architecture community it is traditionally distinguished from "simulator" by what exactly is being reproduced. An emulator reproduces the functional behavior of a processor: given the same instructions and the same inputs, it produces the same outputs, so that real programs written for the target architecture run correctly. A simulator reproduces timing and microarchitectural behavior: pipeline stages, cache hit and miss patterns, branch predictor state, out-of-order issue, retirement latency. The two goals pull in opposite directions.

A tool at the functional-behavior end can take shortcuts a real CPU cannot: it can execute one instruction "in zero time", skip the pipeline entirely and just compute the result, or even recognize a whole pattern of instructions and emit the answer in one step. That is fine because the guest program never measures how long its own code takes to run; it just observes that the answer is right. A cycle-accurate tool has to refuse those shortcuts—the whole point is to tell you how many cycles a real CPU would have spent. It therefore ends up orders of magnitude slower than functional emulation, and is mostly used for compiler back-end tuning, architecture research papers, and evaluating cache designs before committing silicon.

The landscape sits along that axis:

```
tool             kind              used for
--------------   ---------------   --------------------------
\plan 5i         functional        run ARM user programs
QEMU             functional        run whole OSes + user progs
Bochs            functional        debug x86 OS kernels
SPIM             functional        teach MIPS
Dolphin          functional        run GameCube games
gem5             cycle-accurate    architecture research
```

`5i` sits firmly on the functional side: it emulates ARM instructions well enough to run real Plan 9 binaries, but it has no model of pipeline, cache, or memory latency. This is the right trade-off for a debugging and teaching tool—you can single-step a program, watch every instruction fire, and get a trace that is readable rather than lost inside a million pipeline-stage entries. It also keeps the code under 3k lines, because one C `switch` statement can cover the whole ARM ISA.

2.2.2 User-mode vs full-system emulation

The second big design choice an emulator has to make is how much of the target machine to emulate. At one extreme is full-system emulation: the emulator models not just the CPU but also the memory controller, the MMU, the interrupt controller, the disk, the network card, the timer, the video output—enough hardware that a stock OS kernel can boot inside the emulator and never notice it is not running on real silicon. At the other extreme is user-mode emulation: the emulator reproduces only the user-mode CPU, and every time the

guest program issues a system call the emulator catches it and forwards the request to the host OS instead of emulating the kernel that would normally service it.

The full-system route is more faithful but much more work. QEMU's system mode has millions of lines devoted to modelling peripherals, and every new supported machine (each Raspberry Pi variant, each embedded board) takes weeks of per-device engineering. Bochs, Dolphin (GameCube), and RPCS3 (PS3) are all full-system emulators because their goal is to run unmodified kernels or console firmware, for which there is no other choice. The user-mode route is a huge shortcut: no MMU page-walking, no device drivers to emulate, no interrupt dispatch—the "kernel" is whatever kernel is already running on the host, reached through a thin syscall forwarding layer.

Every major ISA emulator offers both modes or has a user-mode cousin:

```
target ISA            user-mode tool    full-system tool
-------------------   ---------------   -----------------
ARM on x86            qemu-arm          qemu-system-arm
x86 on ARM            qemu-i386         qemu-system-i386
x86 on macOS (ARM)    Rosetta 2         (none, uses host)
x86 on Windows ARM    xtajit (WoW64)    Hyper-V
Win32 on Linux        Wine              (API layer only,
                                         no CPU emulation)
\plan ARM on host     5i                (none)
```

The user-mode versions all share one simplifying assumption: the guest program is pure user-space code that only touches the world through the syscall boundary. That covers almost every interesting program but excludes device drivers, kernel modules, and anything that pokes at physical memory directly. Wine is a slightly different animal because it does not emulate the x86 CPU at all—it runs native x86 code on a native x86 host and only translates the Win32 API calls to POSIX equivalents. That is user-mode "emulation" in the same sense as `5i`, just with the CPU layer removed.

`5i` is a pure user-mode emulator and does not pretend otherwise. When the guest ARM binary hits a `SWI` instruction (the ARM system call trap), `5i` does not dispatch it into a simulated kernel—there is no simulated kernel. Instead it reads the guest's registers to recover the syscall number and arguments, copies any pointer arguments out of the guest's simulated memory into real buffers, performs the equivalent host syscall (opening a real file, reading from a real fd, writing to real stdout), and writes the result back into the guest's registers. The "OS" the guest sees is the host Plan 9 kernel, reached through Chapter 7. This is the "cheating" the section-head comment refers to, and it is what keeps the whole emulator under 3k lines: no page tables, no scheduler, no device drivers, no interrupt vectors—just a big syscall-forwarding `switch` statement.

2.2.3 Interpretation styles

Once the scope of emulation is settled (functional or cycle-accurate, user-mode or full-system), the last fundamental choice is how to dispatch from one instruction to the next. The naive loop reads the next instruction, figures out what it is, and calls a handler—but there are several ways to organize that dispatch, and they differ by more than an order of magnitude in speed.

The simplest form is switch dispatch: one C `switch` with one `case` per opcode, inside a `while` loop. Each iteration fetches the next instruction, masks off the opcode bits, and lets the `switch` pick the right handler. This is how `5i` works, and also Bochs, SPIM, and most small teaching emulators. The code looks exactly like you would write it by hand in an afternoon, which is a big part of why it is the starting point for every emulator. The cost is that every instruction pays the price of a computed branch back to the top of the loop plus a switch dispatch, typically ten to fifteen native instructions of overhead per guest instruction, putting throughput around ten times slower than native execution.

Threaded code (also called "computed goto" or "direct-threaded interpretation") replaces the `while`+`switch` with one label per handler and a `goto *next` at the end of each handler. This removes the per-iteration dispatch

overhead—the CPU sees a jump directly from one handler to the next, and its branch predictor can learn common patterns. GCC's `&&label` extension makes this practical in C; CPython's bytecode loop, OCaml's bytecode interpreter, and Lua 5's VM all use the technique, typically landing around three to five times slower than native.

Dynamic binary translation, better known as JIT, abandons instruction-at-a-time dispatch entirely. It reads a whole basic block of guest instructions, translates them into native host instructions once, caches the translation, and then runs native code until the block ends. QEMU's Tiny Code Generator works this way; so do Apple Rosetta 2, Android's ART, HotSpot, V8, and FEX-Emu. JIT gets within 1.1 to 2x of native but is a much larger engineering undertaking, because every guest instruction needs a native-code template and the runtime has to manage translation caches, invalidation on self-modifying code, and signal-handling re-entry in the middle of translated code.

`5i` picks switch dispatch and sticks with it. The whole instruction interpreter is one `for` loop in `run()`[40c] containing an `arm_class()`[41] dispatch that leads to per-class handlers (data processing, memory access, control flow). Given this book's goal of explaining how an emulator works, switch dispatch is the right trade-off: a threaded interpreter would obscure the dispatch in macro tricks, and a JIT would require an entire extra codegen layer worth more than the rest of the book. The slowness does not matter for a teaching and debugging tool where the user is likely to be single-stepping instructions anyway.

2.2.4 Memory model

One last design choice remains: how to store the guest's address space inside the host's. The guest program expects a flat 32-bit virtual address space that it can load and store at any aligned byte offset; the emulator has to make that work using nothing but ordinary host memory. The range of options is surprisingly wide.

The simplest option is a flat array: `char mem[4G]` (or as much as the guest can plausibly use), indexed directly by the guest address. The smallest teaching emulators (SPIM, many undergraduate-course projects) do exactly this. The code is trivial—`mem[addr]` in C—but the model has two serious limitations. First, it commits an enormous host allocation up front even if the guest only touches a tiny fraction of its address space. Second, every address is always valid, so the emulator cannot model a segmentation fault on an unmapped page the way a real program would see one.

`5i` and QEMU's `softmmu` (and Bochs before them) take the next step up: a paged model with a software TLB. The guest address space is split into fixed-size pages; pages are allocated on demand as the guest touches them (producing a real fault when it touches an unmapped one); and a small hash or direct-mapped cache of recent guest-page to host-pointer translations sits in front of the allocator so that the common case (touching a page you touched recently) is a single table lookup and a base-plus-offset load. `5i` builds this out of `page_of_vaddr()`[67], a `Tlb`[69b] struct, and a separate `ifetch()`[70] path that layers an instruction cache on top, all described in Chapter 6.

The fastest option abandons the software TLB entirely and mmaps the guest's address space directly onto the host's, so that guest loads and stores become plain host loads and stores with no software lookup in between. QEMU's Linux user-mode target does this when the host and guest have the same word size, and Apple Rosetta 2 does it throughout, which is part of why Rosetta's overhead is closer to 15% than the 10x of a switch-dispatch interpreter. The cost is that the host and guest must agree on pointer layout (no 64-bit guest on a 32-bit host), the host OS's MMU becomes implicitly part of the emulator, and the guest can no longer be traced through the emulator's own memory handlers for debugging. `5i` gives that speed up in exchange for keeping the whole memory layer under a thousand lines of straightforward C that you can step through.

2.3 `5i` interfaces

I just described the general principles behind emulation. I will now focus on the practical interface of `5i`: how to invoke it from the command line, and how to use its built-in debugger to inspect program execution.

2.3.1 Command-line interface

The command-line interface of `5i` is minimal:

```
5i [5.out] [arg1 arg2 ...]
```

The first argument is the ARM binary to execute (defaulting to `5.out`, the conventional name for ARM executables in Plan 9). Any remaining arguments are passed to the emulated program as its `argv`. Unlike `rc` or `mk`, `5i` takes no flags—there are no options to toggle. The emulator simply loads the binary, initializes memory and the stack, and drops into its debugger prompt.

2.3.2 Debugger interface

Rather than running the program immediately, `5i` starts in an interactive debugger—a simplified version of the Plan 9 debugger `db` (see the DEBUGGER book [Pad16c]). At the prompt, you enter commands to control execution and inspect state. The most important commands are:

- `:c` — continue (run the program until it exits or hits a breakpoint)
- `:s` — single-step one instruction
- `$r` — dump all registers
- `$t` — toggle instruction tracing
- `$q` — quit the emulator
- `?` and `/` — examine memory (text and data respectively)

Pressing RETURN on an empty line repeats the last command, which is especially useful when single-stepping: you type `:s` once, then just keep pressing RETURN.

A particularly useful combination is `$t` followed by `:c`: this runs the program while printing every instruction as it executes, producing a complete execution trace. This is invaluable for debugging—you can see exactly which instructions were executed before a crash, without needing to set breakpoints.

You can also examine specific addresses. For example, `main?i` disassembles the instructions at the address of `main`, while `exit?iii` disassembles three instructions starting at `exit`. The `?` modifier reads from the text segment and `/` from the data segment, following the same convention as `db`.

2.4 helloworld

To make things concrete, here is a minimal `helloworld.s` program in Plan 9 ARM assembly (Asm5):

⟨helloworld.s 20⟩≡

```
TEXT main(SB), $0
    MOVW $1, R0          /* fd = stdout */
    MOVW $str(SB), R1    /* buf */
    MOVW $14, R2         /* count */
    SWI  $0              /* pwrite() syscall */
    RET

DATA str(SB)/14, $"hello, world\n"
GLOBL str(SB), $14
```

This is the same `helloworld.s` discussed in the ASSEMBLER book [Pad15a]. The difference is the perspective: the Assembler book explains how `5a` translates the source text into machine code, while here we care about what happens when `5i` executes it. After assembling with `5a` and linking with `5l`, we can run it under `5i`:

```
% 5a helloworld.s
% 5l -o 5.out helloworld.5
% 5i 5.out
5i
:c
hello, world
%
```

Despite its simplicity, this program exercises most of the principles described in the previous sections. For each instruction, `5i` performs the fetch-decode-execute loop: it reads four bytes from simulated memory at the program counter address, decodes the opcode and operands, and dispatches to the corresponding C function. The three `MOVW` instructions each write an immediate value into a register—`5i` simply stores the value into the appropriate slot of its `reg.r` array. The `SWI` instruction is more interesting: rather than forwarding it to a real ARM interrupt handler, `5i` intercepts it and translates the system call into a host `pwrite()`—this is the user-level emulation strategy described earlier, where the emulator handles system calls by calling the equivalent host OS function. Finally, `RET` sets the program counter to the return address stored in the link register, and the emulator exits when it detects the program has finished.

2.5 Input binary executable

The input to `5i` is a Plan 9`a.out` executable—the same binary that the kernel loader would load on a real ARM machine. This is much simpler than a full-system emulator like QEMU, which takes an entire disk image and must emulate the boot process, BIOS, and device initialization. Because `5i` is a user-level emulator, it only needs the executable itself: an `a.out` header followed by the text (code) and data segments containing ARM machine instructions and initialized data. The `a.out` format is fully described in the LINKER book [Pad15b], where it is the output; here it is the input. The layout is straightforward:

```
+--------------------+
| a.out header       |  magic, sizes, entry point (32 bytes)
+--------------------+
| Text segment       |  ARM instructions
+--------------------+
| Data segment       |  initialized data
+--------------------+
| Symbol table       |  (optional, for debugger)
+--------------------+
```

`5i` uses `libmach`'s `crackhdr()` function to parse the header and extract the segment sizes and entry point, then loads the text and data segments into simulated memory.

2.6 The ARM architecture

ARM is a RISC (Reduced Instruction Set Computer) architecture, originally designed at Acorn Computers in 1985. Compared to x86, it has a small, regular instruction set: all instructions are 32 bits wide, there are 16 general-purpose registers, and memory access is restricted to explicit load/store instructions. ARM has a few distinctive features that the emulator must handle. Every instruction carries a 4-bit condition code field, allowing

conditional execution without branching—a design choice that compensates for the lack of branch prediction in early ARM processors.[1] There is no dedicated `CALL/RET` instruction pair: function calls use `BL` (Branch and Link), which saves the return address in the link register (`R14`), and returning is simply a branch back to that register.

ARM is technically bi-endian—it can operate in either byte order—but in practice (and in Plan 9) it runs little-endian. This matters for the emulator: when `5i` fetches a 32-bit instruction from memory, it reconstructs it as `va[0] | va[1]<<8 | va[2]<<16 | va[3]<<24`, reading the least significant byte first.

The ARM instruction set has a high degree of symmetry: arithmetic operations share a common encoding format with regular bit fields for the opcode, condition code, and operands. This regularity means that `5i`'s instruction decoder is not a giant `switch` with hundreds of cases; instead, it factors out common patterns like the shifter operand and the condition check, reflecting the structure of the hardware itself.

The specific ARM variant emulated by `5i` corresponds roughly to ARMv4, the architecture of the StrongARM processors that Plan 9 originally targeted. This is also close to the ARMv6 used by the original Raspberry Pi, which makes `5i` a useful tool for understanding code running on that platform.

Finally, because `5i` decodes binary instructions by extracting bit fields, its source code is full of C bit manipulation idioms: shifts (`<<`, `>>`), masks (`& 0xf`), and sign extension. These patterns will be a recurring theme throughout this book.

2.7 Code organization

The source code of `5i` is split into about a dozen files, all sharing a common header. Table 2.1 gives an overview.

Function	Ch.	File	Entities
data structures, prototypes	3	`arm.h`	`Registers`[27c] `Memory`[28c] `Icache`[71a]
entry point, loading	4	`5i.c`	`main()`[31e] `initmemory()`[34a] `initstk()`[36]
instruction interpreter	5	`run.c`	`run()`[40c] `armclass()X`
memory emulation	6	`mem.c`	`getmem\_w()X` `putmem\_w()X`
instruction cache	6	`icache.c`	
syscall emulation	7	`syscall.c`	`dosyscall()X`
breakpoints	8	`bpt.c`	`brkchk()`[110a]
symbol lookup	8	`symbols.c`	
debugger commands	8	`cmd.c`	`cmd()`[37c]
profiling/statistics	9	`stats.c`	`iprofile()`[115]
globals	C	`globals.c`	
utilities	C	`utils.c`	
Total			∼3000 LOC

Table 2.1: Source files of `5i`.

[1] The idea is that a short `if/else` can be compiled without any branch instruction: both sides are emitted sequentially, each guarded by opposite condition codes. On a simple pipeline with no branch predictor, this avoids the costly pipeline flush that a mispredicted branch would cause. However, it wastes instruction slots when the condition is false—the processor still fetches and decodes the instruction, only to discard it. As ARM processors grew more sophisticated and acquired good branch predictors, conditional execution became a net loss: it uses up four precious bits in every instruction (the condition field) and prevents out-of-order execution from speculating past the guarded instructions. ARM64 (AArch64) accordingly dropped pervasive conditional execution, keeping only a few conditional select/compare instructions instead.

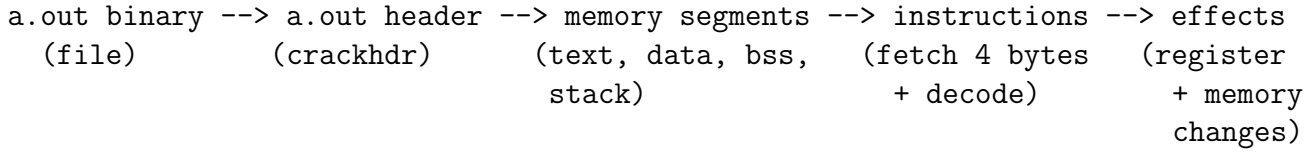


Figure 2.1: Data flow diagram of `5i`.

Every source file includes the same set of headers:

⟨*basic includes* 23⟩≡ (131 130 128 127 126)

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <mach.h>

#include "arm.h"
```

2.8 Software architecture

Figure 2.1 describes the main data flow of `5i`, whereas Figure 2.2 describes the main control flow.

As shown by Figure 2.1, `5i` reads an `a.out` binary file, parses its header with `crackhdr()X` from `libmach`, loads the text and data segments into simulated memory, and then repeatedly fetches 32-bit instructions from that memory. Each instruction is decoded and produces effects on the simulated registers and memory—the fetch-decode-execute loop.

Figure 2.2 shows the control flow starting from `main()`[31e] at the top. After loading the binary (`crackhdr()X`), initializing memory (`initmemory()`[34a]), and building the initial stack (`initstk()`[36]), `main()` enters the debugger command loop (`cmd()`[37c]). When the user types `:c`, the interpreter loop `run()`[40c] takes over, fetching and executing instructions one at a time. Each instruction passes through `armclass()X` for decoding, then dispatches to the appropriate handler (one of many `Iop_xxx` functions). Arithmetic and logic handlers update the register file directly. Load/store handlers go through the memory emulation layer (`getmem_w()`[72c], `putmem_w()`[74b], etc.). The `SWI` instruction triggers `dosyscall()X`, which translates the ARM system call into the equivalent host OS call (open, read, write, fork, exec, etc.).

Compared to the other tools in Principia Softwarica, the architecture of `5i` is remarkably simple. There is no complex parsing stage—unlike the assembler or compiler, which must tokenize and parse source text, the emulator just reads a `ulong` (4 bytes) from memory. The ARM instruction format is regular enough that `armclass()X` can decode it with a few shifts and masks, then dispatch through a function pointer table. Note also that `5i` works with the final binary executable format (`a.out`), not the Plan 9 object format (`5.out`); it has no dependency on `5.out.h`.

2.9 Book structure

You now have enough background to understand the source code of `5i`. The rest of the book is organized as follows. I will start by describing the core data structures of `5i` in Chapter 3: the register file, memory segments, and instruction cache. Then, Chapter 4 presents `main()`[31e], which loads the binary, initializes memory and the stack, and enters the debugger command loop. Chapter 5 is the heart of the book: it describes the instruction interpreter—the fetch-decode-execute loop and the handlers for each ARM instruction class (arithmetic, memory, branches, etc.). Chapter 6 covers the memory emulation layer, and Chapter 7 explains how `5i` intercepts system calls and translates them into host OS calls. Chapter 8 presents the built-in debugger, and Chapter 9 the simple instruction profiler. Finally, the appendices contain debugging support, error management, and utility functions.

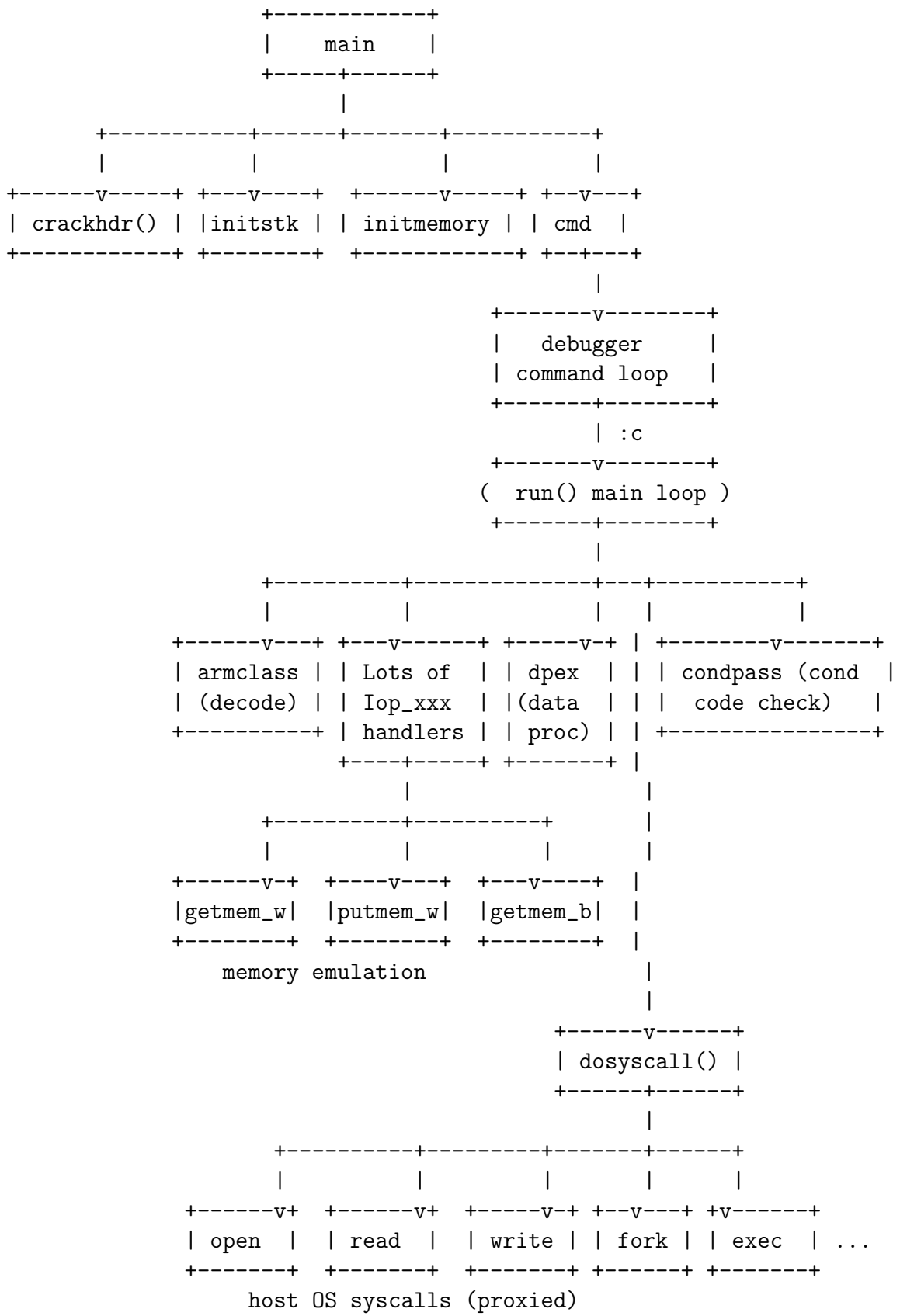


Figure 2.2: Control flow diagram of `5i`.

Chapter 3

Core Data Structures

Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won't usually need your code; it'll be obvious.

Fred Brooks

In this chapter, I will present the core data structures of `5i`. The first two sections describe how `5i` represents ARM instructions: the raw 32-bit `instructionX` type, the `opcode`[26a] enumeration, and the `Inst`[26c] dispatch table that maps opcodes to handler functions. The following sections describe the machine state that `5i` simulates: the `Registers`[27c] structure (the 16 ARM registers) and the `Memory`[28c] structure (the process address space, divided into text, data, BSS, and stack segments).

3.1 `Instruction` and `Opcode`

The ARM uses fixed-length instructions of 4 bytes[1]:

⟨*typedef instruction* 25⟩≡ (124)
```
typedef ulong instruction;
```

This is all that `5i` needs to represent an instruction in memory: a single 32-bit word. The simplicity of this type—compared to the elaborate AST nodes of the compiler or the token structures of the assembler—reflects the fact that `5i` operates on the final, flat binary format. All the structure (opcode, operands, condition code) is encoded as bit fields within this word.

An ARM instruction is composed of an opcode and a few operands.

Here is how the 32 bits of an ARM data-processing instruction (e.g., `ADD`, `SUB`, `AND`) are laid out:

```
31 28 27 26 25 24    21 20 19  16 15  12 11             0
+----+-----+--+-------+--+------+------+----------------+
|cond| 0 0 | I| opcode| S|  Rn  |  Rd  |   operand2     |
+----+-----+--+-------+--+------+------+----------------+
 4b    2b  1b   4b    1b   4b     4b        12 bits
```

The `cond` field is the condition code that makes every ARM instruction conditional. The 4-bit `opcode` selects the operation (ADD, SUB, etc.), `Rn` and `Rd` are the source and destination registers, and `operand2` encodes either an immediate value or a shifted register. The function `armclass()X` in `run.c` extracts these fields with shifts and masks to produce an `opcode`[26a] index. Note that the `opcode` enumeration below does not correspond directly to the 4-bit opcode field in the ARM instruction. In the actual hardware encoding, the opcode is spread

[1]The Plan 9 C compilers define `long` as 32 bits on every platform, including 64-bit architectures—a deliberate choice to keep data structures the same size everywhere. Under GCC or Clang, however, `long` is typically 64 bits on a 64-bit host, so `ulong` would be 8 bytes instead of 4. Ideally, this code should use an explicit `u32int` (or `uint32\_t`) type to avoid any ambiguity.

across several bit fields (the top two bits select the instruction class, then the 4-bit opcode refines it within that class), and different instruction formats reuse the same bit positions for different purposes. The `enum opcode` in `5i` flattens all of this into a single linear index suitable for dispatching through `itab`[27a]:

⟨*enum* `opcode` 26a⟩≡ (124)
```
enum opcode {
    // ----------------------------------------
    // Arithmetic and logic opcodes
    // ----------------------------------------
    ⟨arith/logic opcodes 42⟩
    // ----------------------------------------
    // Memory MOV opcodes
    // ----------------------------------------
    ⟨memory opcodes 55a⟩
    // ----------------------------------------
    // Control flow opcodes
    // ----------------------------------------
    ⟨branching opcodes 62a⟩
    // ----------------------------------------
    // Syscall opcodes
    // ----------------------------------------
    ⟨syscall opcodes 66a⟩

    // for opcodes not handled by 5i
    OUNDEF   = 89
};
```

The opcodes are organized by category—arithmetic/logic, memory, branches, syscalls—mirroring the instruction categories discussed in Chapter 2. The last entry, `OUNDEF`, is a catch-all for ARM instructions that `5i` does not implement (e.g., coprocessor instructions, or opcodes used only by the kernel). If the interpreter encounters one, it reports an error rather than silently producing wrong results.

For profiling purposes, opcodes are also grouped into broader categories:

⟨*enum* `ixxx` 26b⟩≡ (124)
```
enum opcode_category
{
    Iarith,
    Imem,
    Ibranch,
    Isyscall,
    Imisc,
};
```

3.2 `Inst` and `itab`

Each opcode is associated with an `Inst`[26c] structure that bundles the handler function, a human-readable name (for the debugger and profiler), and the broad category:

⟨*struct* `Inst` 26c⟩≡ (124)
```
struct Inst
{
    void  (*func)(instruction);
    char* name;
    // enum<opcode_category>
    int type;

    ⟨Inst profiling fields 111a⟩
};
```

The global array `itab`[27a] maps each `opcode`[26a] value to its `Inst` entry. This is the central dispatch table of the emulator: after `armclass()X` decodes an instruction into an opcode, the interpreter simply calls `itab[opcode].func(instruction)` to execute it.

⟨*global* `itab` 27a⟩≡ (128c)
```
//map<enum<opcode>, Inst>
Inst itab[] =
{
  ⟨itab elements 27b⟩
  { 0 }
};
```

The only entry shown here is `OUNDEF`, which maps to the `undef()`[122d] error handler. The remaining 88 entries (arithmetic, memory, branches, etc.) will be filled in gradually in Chapter 5.

⟨`itab` *elements* 27b⟩≡ (27a) 44b▷
```
[OUNDEF] =  { &undef,  "UNDEF", Imisc},
```
Uses `Imisc` 26b and `undef()` 122d.

3.3 Registers and reg

The set of registers is the core state of the simulated processor. ARM has 16 general-purpose 32-bit registers (`R0`–`R15`), three of which have a special role: `R13` is the stack pointer, `R14` is the link register (return address for `BL`), and `R15` is the program counter.

⟨*struct* `Registers` 27c⟩≡ (124)
```
struct Registers
{
    long r[16];
    ⟨Registers other fields 40b⟩
};
```

⟨*global* `reg` 27d⟩≡ (126b)
```
Registers reg;
```

Here is the logical layout of the simulated CPU state as `5i` grows it across the book—the general-purpose bank plus the bookkeeping fields that `itab` handlers and `run()`[40c] sprinkle into `Registers` as they get introduced:

```
struct Registers
+---------------------------------+
| r[0]   R0     arg0 / return     |  <-- REGARG / REGRET
| r[1]   R1                       |
| ...                             |
| r[12]  R12                      |
| r[13]  R13    stack pointer (SP)|  <-- REGSP
| r[14]  R14    link register (LR)|  <-- REGLINK
| r[15]  R15    program counter(PC)|  <-- REGPC
+---------------------------------+
| cbit          latched carry     |  (ARM C flag, lazy)
| cout          shifter carry out |  (set by shift())
| cc1, cc2      lazy flag inputs  |  (for Z/N/V eval)
| compare_op    CCcmp / CCnone    |
| instr_cond    bits 28..31       |
+---------------------------------+
| ar            internal PC copy  |  } fetched each
```

```
| instr         32-bit word        |  } iteration of
| instr_opcode  itab index         |  } the run() loop
| ip            &itab[opcode]      |  }
+-----------------------------------+
```

The four fields in the bottom group are pipeline-stage snapshots owned by `run()`; they sit inside `Registers` by convention (like hidden pipeline latches in real hardware) but are not visible to the guest program. Real ARM hardware also has a packed `CPSR` word with N/Z/C/V flag bits; `5i` keeps those as separate ints and even defers most flag evaluation until a `Bxx` branch actually needs them—hence `cc1`/`cc2`/`compare_op` described in Section 5.4.6.

The special registers are given symbolic names so the code can say `reg.r[REGPC]` instead of `reg.r[15]`:

⟨*enum* `regxxx` 28a⟩≡ (124)
```
enum
{
    REGARG = 0,
    REGRET = 0,

    REGSP = 13,
    REGLINK = 14,
    REGPC = 15,
};
```

Note that `REGARG` and `REGRET` are both `R0`: in the Plan 9 calling convention, the first argument to a function and its return value share the same register. This is a Plan 9 convention, not an ARM requirement, but `5i` needs to know it because it emulates system calls by reading arguments from (and writing results to) these registers.

3.4 Segment and memory

The memory model of `5i` does not simulate a flat physical address space (0 to 4GB) with page tables and a TLB. Instead, it directly models the process virtual memory as the kernel would set it up after `exec()`: four segments for text (code), data, BSS (uninitialized data), and stack.

⟨*enum* `segment_kind` 28b⟩≡ (124)
```
enum segment_kind
{
    Text,
    Data,
    Bss,
    Stack,

    Nseg,
};
```

⟨*struct* `Memory` 28c⟩≡ (124)
```
struct Memory
{
    //map<enum<segment_kind>, Segment>
    Segment seg[Nseg];
};
```
Uses `Nseg` 28b.

⟨*global* `memory` 28d⟩≡ (126b)
```
Memory memory;
```

Here is how the `Memory` structure maps to a process's virtual address space:

```
High addresses
+-------------------+ <-- STACKTOP
|      Stack        |  memory.seg[Stack]
|  (grows down)     |
+-------------------+ <-- stack base
|                   |
|   (unmapped gap)  |
|                   |
+-------------------+ <-- bss end
|       BSS         |  memory.seg[Bss]
| (zero-initialized)|
+-------------------+ <-- data end = bss base
|      Data         |  memory.seg[Data]
|(initialized from  |
|    a.out file)    |
+-------------------+ <-- text end = data base
|      Text         |  memory.seg[Text]
|  (ARM instructions|
|   from a.out file)|
+-------------------+ <-- UTZERO (text base)
Low addresses
```

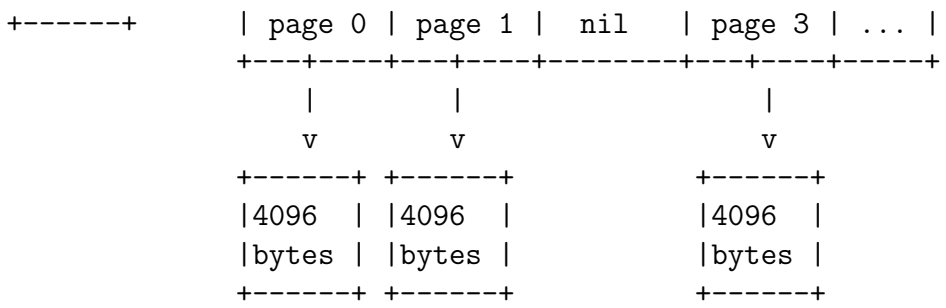
Each segment knows its virtual address range (`base` to `end`) and has a page table—an array of pointers to 4096-byte pages—that holds the actual data. For the text and data segments, the initial content comes from the binary file (hence `fileoff`/`fileend`), and pages are loaded lazily on first access.

⟨*struct* `Segment` 29⟩≡ (124)

```
struct Segment
{
    // enum<segment_kind>
    short type;

    uintptr base;
    uintptr end;

    //array<option<array_4096<byte>>> page table
    byte** table; // the data

    // for the Text and Data segments the bytes are in the file
    ulong fileoff;
    ulong fileend;

    ⟨Segment profiling fields 114c⟩
};
```

The `table` field deserves some explanation. It is a two-level structure: an array of pointers to 4096-byte pages, where each page is allocated on demand (initially `nil`).

```
Segment
+------+
| base |
| end  |
| table+----> +--------+--------+--------+--------+-----+
```



When `getmem_w()`[72c] or `putmem_b()`[73d] accesses an address, it subtracts the segment's `base` to get an offset, divides by 4096 to find the page index, and uses the remainder as the byte offset within the page. A `nil` page pointer means the page has not been loaded yet: for the text and data segments, this triggers lazy loading from the binary file; for BSS and stack, a fresh zero-filled page is allocated. The segfault case is handled differently: `page_of_vaddr()`[67] first searches for a segment whose `base`–`end` range contains the address. If no segment matches (e.g., the address falls in the unmapped gap), `5i` reports a fault—the `nil` page pointers are never reached in that case.

Chapter 4

main()

I now switch from the bottom-up approach of Chapter 3 to a top-down approach; indeed, I will describe in the following chapters the main functions of 5i, starting in this chapter with main()[31e], the entry point of the emulator.

4.1 main() skeleton and globals

The main()[31e] function of 5i performs four tasks: it opens the binary file and parses its a.out header (inithdr()[32b]), it loads the text and data segments into simulated memory (initmemory()[34a]), it builds the initial stack with argc/argv (initstk()[36]), and then it enters the debugger command loop (cmd()[37c]). I will first present the globals used by main(), then the function itself, and then each of the initialization functions in turn.

The default binary name is 5.out, the conventional name for ARM executables produced by 5l:

⟨*global* file 31a⟩≡ (130a)
```
char* file = "5.out";
```
Uses file 31a.

The file descriptor text holds the opened binary. It must be a global because page_of_vaddr()[67] needs it later for lazy loading of text and data pages:

⟨*global* text 31b⟩≡ (126b)
```
fdt text;
```

The emulator uses buffered I/O (Biobuf from libbio) for its debugger input and output:

⟨*global* bxxx 31c⟩≡ (130a)
```
Biobuf bi, bo;
```

⟨*global* bixxx 31d⟩≡ (126b)
```
Biobuf *bin, *bout;
```

The main() function is straightforward:

⟨*function* main 31e⟩≡ (130a)
```
//@Scheck: entry point!
void main(int argc, char **argv)
{

    argc--;
    argv++;

    bout = &bo;
    bin = &bi;
    Binit(bout, STDOUT, OWRITE);
    Binit(bin, STDIN, OREAD);
```

```
    ⟨main() tlb initialisation 69e⟩

    if(argc)
        file = argv[0];
    argc--;
    argv++;

    text = open(file, OREAD);
    if(text < 0)
        fatal(true, "open text '%s'", file);

    Bprint(bout, "5i\n");

    inithdr(text);
    initmemory();
    initstk(argc, argv);

    cmd();
}
```

Uses bi 31c, bin 31d, bo 31c, bout 31d, cmd() 37c, fatal() 122a, file 31a, inithdr() 32b, initmemory() 34a, initstk() 36, and text 31b.

The argument handling is minimal: after skipping `argv[0]` (the emulator's own name), `main()` takes the next argument as the binary filename (defaulting to `5.out`) and shifts `argc`/`argv` again so that the remaining arguments become the emulated program's arguments, which `initstk()` will push onto the simulated stack.

4.2 inithdr()

The `inithdr()`[32b] function uses `libmach`'s `crackhdr()X` to parse the `a.out` header (see the DEBUGGER book [Pad16c] for a full description of `libmach`, and the LINKER book [Pad15b] for the `a.out` format). The parsed information is stored in the global `fhdr`[32a] structure, which `initmemory()`[34a] will read to know the size and offset of each segment:

⟨*global* fhdr 32a⟩≡ (130a)

```
Fhdr fhdr;
```

⟨*function* inithdr 32b⟩≡ (130a)

```
void
inithdr(fdt fd)
{
    ⟨inithdr() locals 33c⟩

    seek(fd, 0, SEEK__START);
    if (!crackhdr(fd, &fhdr))
        fatal(false, "read text header");

    if(fhdr.type != FARM )
        fatal(false, "bad magic number: %d %d", fhdr.type, FARM);

    ⟨inithdr() symmap initialisation 33b⟩
    ⟨inithdr() mach initialisation 33d⟩

}
```

Uses fatal() 122a and fhdr 32a.

After parsing the header, `inithdr()` checks that the magic number corresponds to an ARM executable (`FARM`). It then initializes the symbol table (needed by the debugger for translating addresses to function names) and the `machdata` pointer (needed by `libmach` for architecture-specific operations like disassembly).

⟨*global* `symmap` 33a⟩≡ (126b)
```
Map *symmap;
```

⟨`inithdr()` *symmap initialisation* 33b⟩≡ (32b)
```
if (syminit(fd, &fhdr) < 0)
    fatal(false, "%r\n");

symmap = loadmap(symmap, fd, &fhdr);
```
Uses `fatal()` 122a, `fhdr` 32a, and `symmap` 33a.

⟨`inithdr()` *locals* 33c⟩≡ (32b)
```
Symbol s;
// from libmach.a
extern Machdata armmach;
```

The last step resolves the static base register. In the Plan 9 ARM calling convention, `R12` (called `SB`) serves as a base pointer for accessing global and static data via short offsets (see the ASSEMBLER book [Pad15a] and LINKER book [Pad15b] for how `setR12` is generated and used). The code looks up the symbol `"setR12"` in the binary's symbol table and stores its address in `mach->sb`, so that the debugger can resolve global variable addresses. The `machdata` pointer is set to the ARM-specific function table (`armmach`), which provides architecture-specific operations like disassembly for the debugger commands (see Chapter 8).

⟨`inithdr()` *mach initialisation* 33d⟩≡ (32b)
```
//???
if (mach->sbreg && lookup(0, mach->sbreg, &s))
    mach->sb = s.value;
machdata = &armmach;
```

4.3 `initmemory()`

Now that the header is parsed, `initmemory()`[34a] can set up the simulated address space. This function plays the role of the kernel loader (see the KERNEL book [Pad14] for the real one): it takes the sizes from `fhdr`[32a] and creates the four segments described in Chapter 3. First, a few constants:

⟨*enum* `_anon_ (machine/5i/arm.h)` ◁ 33e⟩≡ (124)
```
enum
{
    /* Plan9 Kernel constants */
    BY2PG  = 4096,
    BY2WD  = 4,

    UTZERO  = 0x1000,
    STACKTOP = 0x80000000,
    STACKSIZE = 0x10000,

    ⟨constant PROFGRAN 112b⟩
    ⟨constant Sbit 47b⟩
    ⟨constant SIGNBIT 65a⟩
};
```

`BY2PG` (bytes per page, 4096) and `BY2WD` (bytes per word, 4) are standard Plan 9 kernel constants. `UTZERO` (`0x1000`) is the base address where user programs are loaded—the first page is left unmapped so that null pointer dereferences cause a fault. `STACKTOP` (`0x80000000`, the 2GB mark) is the top of the user address space, and `STACKSIZE` (`0x10000`, 64KB) is the fixed stack size that `5i` allocates.

The function first computes page-aligned boundaries for each segment by rounding up sizes to the next multiple of `BY2PG` (4096). The idiom `(x + BY2PG-1) & (BY2PG-1)` works because `BY2PG` is a power of two: adding `BY2PG-1` ensures any non-aligned value crosses the next page boundary, and the mask `& (BY2PG-1)` clears the low 12 bits to snap back to a page start (for example, 5000 becomes $(5000 + 4095)$ & $\sim 4095 =$ 9095 & `0xFFFFF000` = 8192, i.e., two pages). After computing these boundaries, the function fills in the `Segment`[29] structures and sets the program counter to the entry point from the header:

⟨*function* `initmemory` 34a⟩≡ (130a)
```
void
initmemory(void)
{
    uintptr t, d, b, bssend;
    ⟨initmemory() locals 34b⟩

    t = (fhdr.txtaddr+fhdr.txtsz+(BY2PG-1)) & ~(BY2PG-1);
    d = (t + fhdr.datsz + (BY2PG-1)) & ~(BY2PG-1);
    bssend = t + fhdr.datsz + fhdr.bsssz;
    b = (bssend + (BY2PG-1)) & ~(BY2PG-1);

    ⟨initmemory() Text segment initilisation 34c⟩
    ⟨initmemory() Data segment initilisation 35a⟩
    ⟨initmemory() Bss segment initilisation 35d⟩
    ⟨initmemory() Stack segment initilisation 35e⟩

    reg.r[REGPC] = fhdr.entry;
}
```
Uses `BY2PG` 33e, `REGPC` 28a, `fhdr` 32a, and `reg` 27d.

⟨`initmemory()` *locals* 34b⟩≡ (34a)
```
Segment *s;
```

Each segment is initialized the same way: set the type, the virtual address range (`base`/`end`), the file offsets (for text and data), and allocate the page table. The text segment comes first:

⟨`initmemory()` *Text segment initilisation* 34c⟩≡ (34a) 34e▷
```
s = &memory.seg[Text];
s->type = Text;
s->base = fhdr.txtaddr - fhdr.hdrsz;
s->end = t;
s->fileoff = fhdr.txtoff - fhdr.hdrsz;
s->fileend = s->fileoff + fhdr.txtsz;
s->table = emalloc(((s->end - s->base)/BY2PG) * sizeof(byte*));
```
Uses `BY2PG` 33e, `Text` 28b, `emalloc()` 123a, `fhdr` 32a, and `memory` 28d.

⟨*global* `textbase` 34d⟩≡ (126b)
```
uintptr textbase;
```

⟨`initmemory()` *Text segment initilisation* 34e⟩+≡ (34a) ◁34c 112c▷
```
textbase = s->base;
```
Uses `textbase` 34d.

The data segment follows immediately after text:

⟨initmemory() *Data segment initilisation* 35a⟩≡ (34a) 35c▷
```
s = &memory.seg[Data];
s->type = Data;
s->base = t;
s->end = t+(d-t);
s->fileoff = fhdr.datoff;
s->fileend = s->fileoff + fhdr.datsz;
s->table = emalloc(((s->end - s->base)/BY2PG)*sizeof(byte*));
```
Uses BY2PG 33e, Data 28b, emalloc() 123a, fhdr 32a, and memory 28d.

⟨*global* datasize 35b⟩≡ (126b)
```
int datasize;
```

⟨initmemory() *Data segment initilisation* 35c⟩+≡ (34a) ◁35a
```
datasize = fhdr.datsz;
```
Uses datasize 35b and fhdr 32a.

BSS follows data. Unlike text and data, it has no file content—pages will be zero-filled on first access:

⟨initmemory() *Bss segment initilisation* 35d⟩≡ (34a)
```
s = &memory.seg[Bss];
s->type = Bss;
s->base = d;
s->end = d+(b-d);
s->table = emalloc(((s->end - s->base)/BY2PG) * sizeof(byte*));
```
Uses BY2PG 33e, Bss 28b, emalloc() 123a, and memory 28d.

The stack is placed at the top of the address space, at a fixed location:

⟨initmemory() *Stack segment initilisation* 35e⟩≡ (34a)
```
s = &memory.seg[Stack];
s->type = Stack;
s->base = STACKTOP-STACKSIZE;
s->end = STACKTOP;
s->table = emalloc(((s->end - s->base)/BY2PG) * sizeof(byte*));
```
Uses BY2PG 33e, STACKSIZE 33e, STACKTOP 33e, Stack 28b, emalloc() 123a, and memory 28d.

4.4 initstk()

With the four memory segments in place, the last step before entering the debugger is to populate the stack. The initstk()[36] function builds the initial stack layout that a Plan 9 process expects after exec(). It pushes argc, the argv pointers, and the argument strings into simulated memory, mimicking what the kernel would do on a real machine. This is another aspect of the user-level emulation: 5i must set up the process environment that the emulated program relies on. For 5i helloworld 5.out foo bar, the stack looks like:

```
STACKTOP  +------------------+
          |       Tos        |  (process info, pid, etc.)
     tos  +------------------+
          | "bar\0"          |  argument strings
          | "foo\0"          |  (packed sequentially)
          | "helloworld\0"   |  argv[0] = program name
       ap +------------------+
          | 0                |  argv terminator (NULL)
          | ptr to "bar"     |  argv[3]
          | ptr to "foo"     |  argv[2]
```

```
            | ptr to "hworld"   |  argv[1] = argv[0]
            +-------------------+
            | argc (= 3)        |
         sp +-------------------+
            |                   |
            v  (stack grows down)
```

⟨*function* initstk 36⟩≡ (130a)

```
void
initstk(int argc, char *argv[])
{
    ulong size;
    ulong sp, ap, tos;
    int i;
    char *p;

    tos = STACKTOP - sizeof(Tos)*2; /* we'll assume twice the host's is big enough */
    sp = tos;
    for (i = 0; i < sizeof(Tos)*2; i++)
        putmem_b(tos + i, 0);

    /*
     * pid is second word from end of tos and needs to be set for nsec().
     * we know arm is a 32-bit cpu, so we'll assume knowledge of the Tos
     * struct for now, and use our pid.
     */
    putmem_w(tos + 4*4 + 2*sizeof(ulong) + 3*sizeof(uvlong), getpid());

    /* Build exec stack */
    size = strlen(file)+1+BY2WD+BY2WD+BY2WD;
    for(i = 0; i < argc; i++)
        size += strlen(argv[i])+BY2WD+1;

    sp -= size;
    sp &= ~7;

    reg.r[0] = tos;
    reg.r[REGSP] = sp;
    reg.r[1] = STACKTOP-4; /* Plan 9 profiling clock (why & why in R1?) */

    /* Push argc */
    putmem_w(sp, argc+1);
    sp += BY2WD;

    /* Compute sizeof(argv) and push argv[0] */
    ap = sp+((argc+1)*BY2WD)+BY2WD;
    putmem_w(sp, ap);
    sp += BY2WD;

    /* Build argv[0] string into stack */
    for(p = file; *p; p++)
        putmem_b(ap++, *p);

    putmem_b(ap++, '\0');

    /* Loop through pushing the arguments */
    for(i = 0; i < argc; i++) {
        putmem_w(sp, ap);
        sp += BY2WD;
```

```
        for(p = argv[i]; *p; p++)
            putmem_b(ap++, *p);
        putmem_b(ap++, '\0');
    }
    /* Null terminate argv */
    putmem_w(sp, 0);

}
```

Uses BY2WD 33e, REGSP 28a, STACKTOP 33e, file 31a, putmem_b() 73d, putmem_w() 74b, and reg 27d.

4.5 cmd()

At this point, the binary is loaded, memory is set up, and the stack is ready. The emulated ARM processor could start running. But instead of executing immediately, main() calls cmd()[37c], which enters the debugger command loop. The bulk of this function is described in Chapter 8; here I only show the global it uses and a skeleton of its structure.

The global dot tracks the "current address" in the debugger, initialized to the program counter. The name comes from the ed editor (see EDITOR book [Pad15c]), where . means "current line"; the debugger db reuses the same convention for "current address":

⟨*global* dot 37a⟩≡ (126b)

```
uintptr dot;
```

⟨cmd() *locals* 37b⟩≡ (37c) 91a▷

```
char *p;
```

⟨*function* cmd 37c⟩≡ (131a)

```
void
cmd(void)
{
    ⟨cmd() locals 37b⟩

    dot = reg.r[REGPC];

    ⟨cmd() initialisation 116⟩

    for(;;) {
        Bflush(bout);
        ⟨cmd() read and parse command and address from user input 91e⟩

        switch(*p) {
        ⟨cmd() command cases 37d⟩
        default:
            Bprint(bout, "?\n");
            break;
        }
    }
}
```

Uses REGPC 28a, bout 31d, dot 37a, and reg 27d.

⟨cmd() *command cases* 37d⟩≡ (37c) 92b▷

```
case ':':
    colon(addr, p+1);
    break;
```

Uses colon() 38a.

I show `colon()`[38a] here because it contains the `:c` (continue) command that actually starts execution—without it, the emulator would just sit in the debugger prompt. The colon commands handle flow control: `:c` to run, `:s` to single-step, `:r` to restart, etc. The full set of colon commands is described in Chapter 8.

⟨*function* `colon` 38a⟩≡ (131a)

```
void
colon(char *addr, char *cp)
{
    ⟨colon() locals 97a⟩

    cp = nextc(cp);

    switch(*cp) {
    ⟨colon() command which return cases 97c⟩
    /* These fall through to print the stopped address */
    ⟨colon() command cases 38b⟩
    default:
        Bprint(bout, "?\n");
        return;
    }

    dot = reg.r[REGPC];

    Bprint(bout, "%s at #%lux ", atbpt? "breakpoint": "stopped", dot);
    ⟨colon() print current instruction 97b⟩
    Bprint(bout, "\n");
}
```

Uses `REGPC` 28a, `atbpt` 108c, `bout` 31d, `dot` 37a, `nextc()` 92a, and `reg` 27d.

And here is the `:c` case, where we finally see the call to `run()`[40c]—the fetch-decode-execute loop that is the heart of the emulator (Chapter 5). The global `count` controls how many instructions to execute; setting it to 0 is a trick: `run()` uses a `do \ldots while(--count)` loop, so 0 wraps to a negative value and the loop runs effectively forever (until a breakpoint or syscall stops it). The `atbpt` flag is cleared so that the "stopped at breakpoint" message is not printed unless an actual breakpoint is hit during execution.

⟨`colon()` *command cases* 38b⟩≡ (38a) 98c▷

```
case 'c':
    count = 0;
    atbpt = false;
    run();
    break;
```

Uses `atbpt` 108c, `count` 40a, and `run()` 40c.

Chapter 5

Instruction Interpreter

This chapter describes the heart of `5i`: the instruction interpreter. It begins with the binary format of ARM instructions, then presents the fetch-decode-execute loop (`run()`[40c]) and the decoder (`arm_class()`[41]), and then covers each category of instructions: arithmetic and logic, memory access (load/store), control flow (branches), and conditional execution.

5.1 Instruction binary format

Every ARM instruction is exactly 4 bytes, with a regular structure: the top 4 bits encode the condition code (Section 5.6.5), the next 3 bits select the instruction "class" (the addressing mode: register/register, immediate/register, etc.), and the remaining bits encode opcodes, register numbers (`Rd`, `Rn`, `Rm`, `Rs`), shift amounts, and immediates. For example, a data-processing instruction like `ADD R2, R3, R1` (R1 = R2 + R3 in Plan 9 syntax) is encoded as:

```
31  28 27 25 24 21 20 19 16 15 12 11         4 3   0
+------+-----+------+--+------+------+--------+------+
| cond |class| opcode| S|  Rn  |  Rd  | shift  |  Rm  |
+------+-----+------+--+------+------+--------+------+
| 1110 | 000 | 0100  | 0| 0010 | 0001 | 00000000| 0011 |
+------+-----+------+--+------+------+--------+------+
  AL    r,r,r   ADD   -   R2     R1    no shift    R3

= 0xE0821003

cond:   condition code (0xe = always)
class:  000 = register/register    (arm_class)
opcode: 0100 = ADD                 (dpex)
S:      set condition flags?
Rn:     first source register
Rd:     destination register
shift:  barrel shifter control     (shift)
Rm:     second source register
```

The ARM ISA has a lot of symmetry. The code in `5i` exploits this by factoring the interpretation along several axes: `arm_class()`[41] is the full decoder—it starts from the 3-bit class field (bits 25–27) but refines further within each class to produce a final opcode index into `itab`[27a], `dpex()`[46d] dispatches on the arithmetic/logic opcode (bits 21–24: AND, ADD, ORR, ...), `shift()`[50b] handles the barrel shifter operand, and the `S` bit is tested ad hoc to decide whether to update condition flags.

5.2 run()

The `run()`[40c] function is the fetch-decode-execute loop. It runs for `count` instructions (as explained in the `:c` case above, `count` is set to 0 for continuous execution).

⟨*global* `count` 40a⟩≡ (126b)
```
int count;
```

The loop uses four internal registers stored in `reg` that mirror the stages of the pipeline: `ar` holds the current program counter, `instr` holds the fetched 32-bit instruction word, `instr_opcode` holds the decoded opcode index (as returned by `arm_class()`[41]), and `ip` points to the corresponding `Inst`[26c] entry in the dispatch table. These are not architectural ARM registers visible to the program, but internal bookkeeping for the interpreter (analogous to hidden pipeline registers in real hardware):

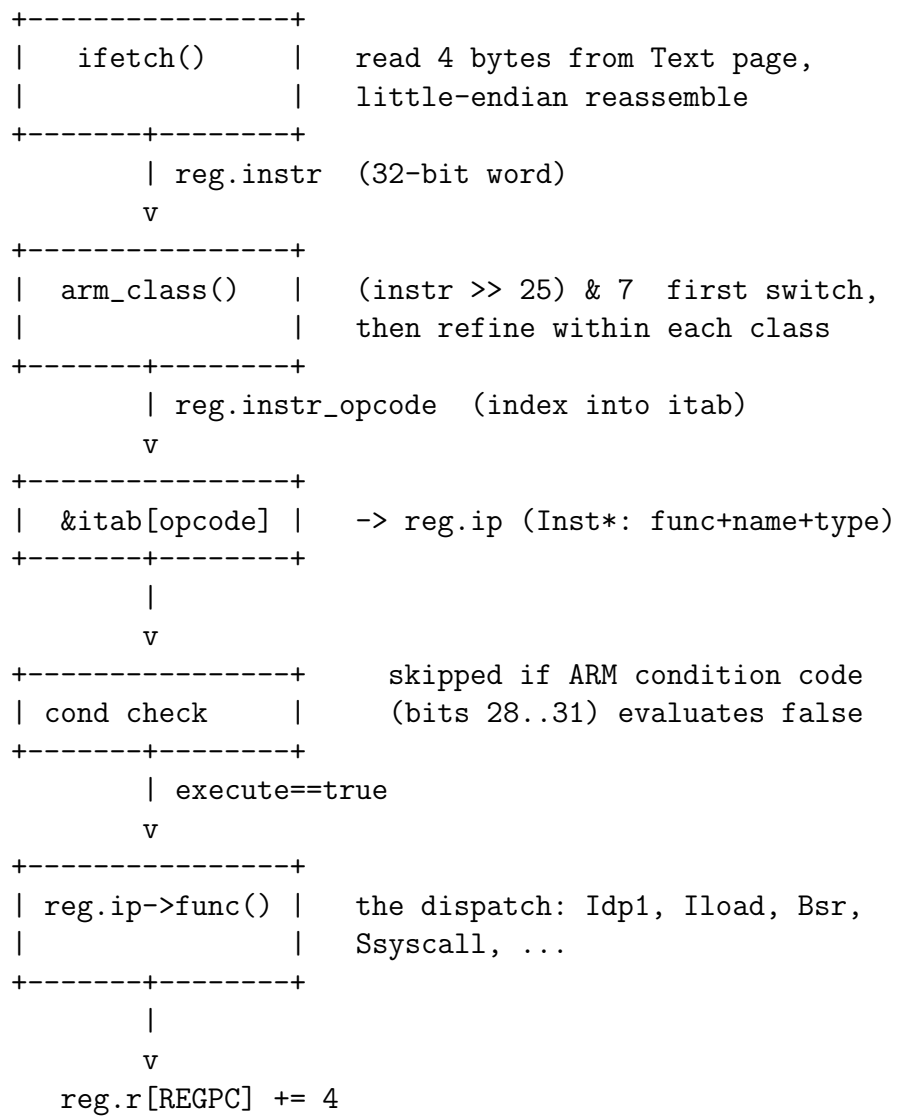
⟨`Registers` *other fields* 40b⟩≡ (27c) 47d▷
```
uintptr  ar;    // reg.r[REGPC]

instruction instr;    // ifetch(reg.ar)
//enum<opcode>
int   instr_opcode; // arm_class(reg.instr)
Inst*  ip;    // &itab[reg.instr_opcode]
```

The loop itself is compact—the entire emulator fits in a few lines:

⟨*function* `run` 40c⟩≡ (128c)
```
void
run(void)
{
    bool execute;

    do {
        reg.ar = reg.r[REGPC];
        reg.instr = ifetch(reg.ar);

        reg.instr_opcode = arm_class(reg.instr);
        reg.ip = &itab[reg.instr_opcode];

        ⟨run() set reg.cond 64b⟩
        ⟨run() switch reg.compare_op to set execute 64e⟩

        if(execute) {
            ⟨run() profile current instruction class 111b⟩
            // !!the dispatch!!
            (*reg.ip->func)(reg.instr);

        }
        else
          if(trace) itrace("%s%s IGNORED",reg.ip->name, cond[reg.instr_cond]);

        reg.r[REGPC] += 4; // simple archi with fixed-length instruction :)

        ⟨run() check for breakpoints 110b⟩
    } while(--count);
}
```
Uses `REGPC` 28a, `arm_class()` 41, `cond-4` 120d, `count` 40a, `ifetch()` 70, `itab` 27a, `itrace()` 120b, `reg` 27d, and `trace` 120a.

Visually, each iteration walks a 32-bit instruction word through four stages before advancing the PC:

```
reg.ar = reg.r[REGPC]
      |
      v
```



Each iteration: fetch the instruction word from memory (`ifetch()`[70]), decode it into an opcode index (`arm_class()`), check the ARM condition code to decide whether to execute, and if so, dispatch through the `itab` function pointer. The program counter always advances by 4 (fixed-length instructions), even for branches—branch handlers compensate by subtracting 4 from `REGPC` so the net effect is correct. I will now describe `arm_class()`, then the individual instruction handlers, and finally the conditional execution mechanism.

5.3 arm_class()

The `arm_class()`[41] function is the decoder: given a 32-bit instruction word, it returns an index into `itab`[27a]. Despite its name, it does more than extract the ARM "class" (bits 25–27): it uses the class as a first-level switch, then examines additional bits within each case to produce the final opcode index (e.g., distinguishing `ADD` from `SUB`, or word loads from byte loads). The code mirrors the structure of `libmach/5db.c` (the ARM disassembler), but simplified since `5i` only needs to handle executable instructions:

⟨*function* `arm_class` 41⟩≡ (128c)

```
int
arm_class(instruction w)
{
    // between 0 and 7
    int class;
    // enum<opcode>
```

```
    int op;
    ⟨arm_class() locals 43a⟩

    class = (w >> 25) & 0x7;
    switch(class) {
    ⟨arm_class() class cases 43b⟩
    default:
        op = OUNDEF;
        break;
    }
    return op;
}
```

Uses OUNDEF 26a.

The 3-bit class field selects one of eight top-level groups. Each `case` of the outer switch further refines using other bits of the instruction to land on a single `itab` entry:

class	bits 27..25	meaning	refiner
0	000	data-processing (r,r,r)	bits 4..7 + 21..24
1	001	data-processing (imm)	bits 21..24
2	010	load/store (imm offset)	bits 20..22
3	011	load/store (reg offset)	bits 20..22
4	100	load/store multiple	bit 20 (L)
5	101	branch / branch-and-link	bit 24 (L)
6	110	coprocessor LDC/STC	-> OUNDEF
7	111	SWI / coprocessor	bit 25 -> OSWI

So the decoder is a two-level dispatch: 3 bits for "what kind of instruction" then a handful more to distinguish the exact operation within that kind. The outer switch is big enough to fit in an 8-entry jump table and the inner refinements are shallow—one reason `arm_class()` stays under a hundred lines for the whole ARM-v5 ISA. Real ARM decoders do essentially the same thing with combinational logic: fanning out bits 25–27 into wordlines that enable per-class sub-decoders.

5.4 Arithmetic and logic

With the decode loop and `arm_class()`[41] in place, I now describe the instruction handlers themselves. Each handler follows the same pattern: extract register numbers and operands from the instruction word using bit shifts and masks, perform the operation, and optionally update the condition flags. I start with arithmetic and logic because they form the bulk of the instruction set.

5.4.1 Opcode extraction

The 16 data-processing opcodes (AND, EOR, SUB, ..., MVN) are numbered 0–15, matching their encoding in bits 21–24 of the instruction word. For each opcode, there are four addressing mode variants, laid out as consecutive blocks of 16 in `itab`: `CARITH0` = 0 (register/register), `CARITH1` = 16 (shifted register), `CARITH2` = 32 (register-shifted register), and `CARITH3` = 48 (immediate). So for example `OADD` (= 4) with a shifted operand becomes `OADD + CARITH1` = 4 + 16 = 20, which indexes directly into the `Idp1` entry in `itab`. This $16 \times 4 = 64$ block fills indices 0–63. Multiplication (`OMUL` = 64) starts right after, since it does not follow the same data-processing pattern and needs its own handler:

⟨*arith/logic opcodes* 42⟩≡ (26a) 43c▷

```
OAND = 0,
```

```
  OEOR = 1,
  OSUB = 2,
  ORSB = 3,
  OADD = 4,
  OADC = 5,
  OSBC = 6,
  ORSC = 7,
  OTST = 8,
  OTEQ = 9,
  OCMP = 10,
  OCMN = 11,
  OORR = 12,
  OMOV = 13,
  OBIC = 14,
  OMVN = 15,
```

⟨arm_class() *locals* 43a⟩≡ (41)
```
  int x;
```

⟨arm_class() *class cases* 43b⟩≡ (41) 52a▷
```
  case 0: /* data processing r,r,r */
      x = ((w >> 4) & 0xf);
      ⟨arm_class() class 0, if x is 0x9 43d⟩
      ⟨arm_class() class 0, if x has 0x9 bits 55f⟩
      // else

      // the opcode! OAND/OADD/...
      op = (w >> 21) & 0xf;

      ⟨arm_class() class 0, adjust op for operands mode 43f⟩
      break;
```

⟨*arith/logic opcodes* 43c⟩+≡ (26a) ◁42 43e▷
```
  OMUL    = 64,
  OMULA   = 65,
```

⟨arm_class() *class 0, if x is 0x9* 43d⟩≡ (43b)
```
  /* mul, swp, mull */
  if(x == 0x9) {
      op = CMUL;
      ⟨arm_class() class 0, when x == 0x9, if bit 24 set 55h⟩
      ⟨arm_class() class 0, when x == 0x9, if bit 23 set 53b⟩
      if(w & (1<<21))
          op++;  /* mla */
      break;
  }
```
Uses CMUL 43e.

⟨*arith/logic opcodes* 43e⟩+≡ (26a) ◁43c 44a▷
```
  CMUL    = 64,
```

⟨arm_class() *class 0, adjust op for operands mode* 43f⟩≡ (43b)
```
  if(w & (1<<4))
    op += CARITH2;
  else
   if((w & (31<<7)) || (w & (1<<5)))
    op += CARITH1;
  // else op += CARITH0
```
Uses CARITH1 44a and CARITH2 44a.

⟨*arith/logic opcodes* 44a⟩+≡ (26a) ◁43e 52b▷
```
CARITH0 = 0,  // r,r,r
CARITH1 = 16, // r<>#, r, r
CARITH2 = 32, // r<>r, r, r
```

5.4.2 Multiplication

I start with multiplication rather than the general data-processing instructions because `Imul`[44c] covers a single operation, making it the simplest example of the handler pattern: extract register fields, perform the operation, done. The general `Idp/dpex` machinery that follows is more involved because it must handle 16 different opcodes across multiple addressing modes.

⟨`itab` *elements* 44b⟩+≡ (27a) ◁27b 45a▷
```
[OMUL]    =  { Imul, "MUL", Iarith },
[OMULA]   =  { Imula, "MULA", Iarith },
```
Uses `Iarith` 26b, `Imul()` 44c, and `Imula()` 44d.

⟨*function* `Imul` 44c⟩≡ (128c)
```
void
Imul(instruction inst)
{
    int rs, rd, rm;

    rd = (inst>>16) & 0xf;
    rs = (inst>>8) & 0xf;
    rm = inst & 0xf;

    if(rd == REGPC || rs == REGPC || rm == REGPC || rd == rm)
        undef(inst);

    reg.r[rd] = reg.r[rm]*reg.r[rs];

    ⟨Imul() trace ??⟩
}
```
Uses `REGPC` 28a, `reg` 27d, and `undef()` 122d.

Note how `Imul` writes its result directly into the global `reg.r[rd]` by side effect—there is no return value. All instruction handlers work this way: they read from and write to the global `reg` structure. Also note that `Imul` rejects `REGPC` as any operand (calling `undef()`[122d]), since it makes no sense to jump to an address that is the result of a multiplication. The general data-processing handlers (`Idp0`[45b] etc.) will need to handle `REGPC` specially, as we will see shortly.

`MLA` (multiply and accumulate) adds a third source register: `Rd = Rm * Rs + Rn`. This is useful for array indexing, where you need a base address plus an index multiplied by the element size:

⟨*function* `Imula` 44d⟩≡ (128c)
```
void
Imula(instruction inst)
{
    int rs, rd, rm, rn;

    rd = (inst>>16) & 0xf;
    rn = (inst>>12) & 0xf;
    rs = (inst>>8) & 0xf;
    rm = inst & 0xf;

    if(rd == REGPC || rn == REGPC || rs == REGPC || rm == REGPC || rd == rm)
        undef(inst);
```

```
        reg.r[rd] = reg.r[rm]*reg.r[rs] + reg.r[rn];

        ⟨Imula() trace ??⟩
    }
```

Uses REGPC 28a, reg 27d, and undef() 122d.

5.4.3 Idp() and dpex()

The three Idp functions (Idp0[45b], Idp1[48d], Idp2[49b]) handle the three addressing mode variants. They all extract the register operands, compute o1 and o2 (applying a shift for Idp1/Idp2), then call dpex()[46d] which switches on the actual arithmetic opcode to perform the operation. This factoring keeps the code compact: $16 \times 3 = 48$ itab entries map to just 3 Idp functions plus one shared dpex().

⟨itab *elements* 45a⟩+≡ (27a) ◁44b 48c▷

```
    //r,r,r
    [OAND] =  { Idp0,  "AND", Iarith },
    [OEOR] =  { Idp0,  "EOR", Iarith },
    [OSUB] =  { Idp0,  "SUB", Iarith },
    [ORSB] =  { Idp0,  "RSB", Iarith },
    [OADD] =  { Idp0,  "ADD", Iarith },
    [OADC] =  { Idp0,  "ADC", Iarith },
    [OSBC] =  { Idp0,  "SBC", Iarith },
    [ORSC] =  { Idp0,  "RSC", Iarith },
    [OTST] =  { Idp0,  "TST", Iarith },
    [OTEQ] =  { Idp0,  "TEQ", Iarith },
    [OCMP] =  { Idp0,  "CMP", Iarith },
    [OCMN] =  { Idp0,  "CMN", Iarith },
    [OORR] =  { Idp0,  "ORR", Iarith },
    [OMOV] =  { Idp0,  "MOV", Iarith },
    [OBIC] =  { Idp0,  "BIC", Iarith },
    [OMVN] =  { Idp0,  "MVN", Iarith },
```

Uses Iarith 26b and Idp0() 45b.

⟨*function* Idp0 45b⟩≡ (128c)

```
    /*
     * data processing instruction R,R,R
     */
    void
    Idp0(instruction inst)
    {
        int rn, rd, rm;
        long o1, o2;

        rn = (inst>>16) & 0xf;
        rd = (inst>>12) & 0xf;
        rm = inst & 0xf;

        o1 = reg.r[rn];
        ⟨adjust o1 if rn is REGPC 46a⟩
        o2 = reg.r[rm];
        ⟨adjust o2 if rm is REGPC 46b⟩

        dpex(inst, o1, o2, rd);

        ⟨Idp0() trace ??⟩
        ⟨Idpx() compensate REGPC 46c⟩
    }
```

Uses dpex() 46d and reg 27d.

Unlike `Imul`[44c], the `Idp` handlers allow `REGPC` as a source or destination, which requires two compensations for the ARM pipeline. When reading `REGPC` as a source, the handler adds 8[1]. When writing `REGPC` as a destination (e.g., `MOV R14, R15` for a return), the handler subtracts 4 to compensate for the `REGPC += 4` at the end of `run()`[40c]:

⟨*adjust o1 if rn is REGPC* 46a⟩≡ (52d 49b 48d 45b)
```
if(rn == REGPC)
    o1 += 8;
```
Uses `REGPC` 28a.

⟨*adjust o2 if rm is REGPC* 46b⟩≡ (49b 48d 45b)
```
if(rm == REGPC)
    o2 += 8;
```
Uses `REGPC` 28a.

⟨`Idpx()` *compensate REGPC* 46c⟩≡ (52d 49b 48d 45b)
```
if(rd == REGPC)
    reg.r[rd] -= 4;
```
Uses `REGPC` 28a and `reg` 27d.

The `dpex()` function ("data processing execute") is where the actual arithmetic or logic operation happens. It takes the two operands `o1` and `o2` (already extracted and shifted by the caller) and switches on the 4-bit opcode to perform the right operation:

⟨*function* `dpex` 46d⟩≡ (128c)
```
void
dpex(instruction inst, long o1, long o2, int rd)
{
    bool cbit = false;

    switch((inst>>21) & 0xf) {
    ⟨dpex() switch arith/logic opcode cases 46e⟩
    }
    ⟨dpex() if Sbit 47c⟩
}
```

5.4.4 Boolean logic

We are now ready to see the concrete execution of each operation. The following subsections fill in the cases of the `dpex()` switch introduced above. I start with the boolean operations, which are the simplest—each is a one-line C expression:

⟨`dpex()` *switch arith/logic opcode cases* 46e⟩≡ (46d) 47a▷
```
case  OAND:
    reg.r[rd] = o1 & o2;
    cbit = true;
    break;
case OORR:
    reg.r[rd] = o1 | o2;
    cbit = true;
    break;
case OEOR:
    reg.r[rd] = o1 ^ o2;
    cbit = true;
    break;
```
Uses `OAND` 42, `OEOR` 42, `OORR` 42, and `reg` 27d.

[1] On real ARM hardware, the 3-stage pipeline (fetch, decode, execute) means that by the time an instruction executes, the PC has advanced two steps, so reading it yields the current instruction's address +8. In `5i`, `reg.r[REGPC]` still holds the current instruction's address (the `+= 4` happens at the end of the loop), so the handler must add 8 explicitly to match what the compiled code expects.

⟨dpex() *switch arith/logic opcode cases* 47a⟩+≡ (46d) ◁46e 48a▷

```
case OBIC:
    reg.r[rd] = o1 & ~o2;
    cbit = true;
    break;
```

Uses OBIC 42 and reg 27d.

As mentioned in the introduction, there is something slightly circular about using C's & operator to explain the ARM AND instruction—after all, & is itself compiled down to an AND instruction. But this is the nature of an emulator written in a high-level language: the host's operations give meaning to the target's operations.

5.4.5 S bit

In ARM, arithmetic instructions only update the condition flags (zero, negative, carry, overflow) when the S bit (bit 20) is set—written ADD.S in 5a syntax (or ADDS in standard ARM syntax) vs. plain ADD. This gives the compiler fine-grained control over flag updates and is what makes conditional execution (Section 5.6.5) practical. Consider this example:

```
CMP    R0, #10        @ sets flags (R0 - 10)
ADD    R1, R2, R3     @ R3 = R1 + R2, does NOT touch flags
MOV.GT R3, R1         @ if R0 > 10, R1 = R3
```

CMP (Section 5.6.4) is really SUB with the S bit always set and the result discarded—it only sets the flags. The key is that the plain ADD on the next line does *not* disturb the flags, so the conditional MOV.GT still tests the result of the CMP and can use the value computed by ADD. On x86, every arithmetic instruction sets the flags, so you cannot interleave computation between a comparison and a conditional instruction this way.

⟨*constant* Sbit 47b⟩≡ (33e)

```
Sbit  = 1<<20,
```

⟨dpex() *if Sbit* 47c⟩≡ (46d)

```
if(inst & Sbit) {
    if(cbit)
        reg.cbit = reg.cout;
    reg.cc1 = reg.r[rd];
    reg.cc2 = 0;
    reg.compare_op = CCcmp;
}
```

Uses CCcmp 64d, Sbit 47b, and reg 27d.

⟨Registers *other fields* 47d⟩+≡ (27c) ◁40b 64a▷

```
int cbit; // carry bit?
int cout;
```

5.4.6 Addition and overflow

Addition is more complex than boolean operations because of the carry flag. As explained in Chapter 2, the same ADD instruction works for both signed and unsigned integers thanks to two's complement, but overflow detection differs between the two interpretations. The carry bit detects unsigned overflow: the XCAST macro

promotes both operands to 64 bits before adding, and if bit 32 of the result is set, the addition carried past 2^{32}. Like the other operations, the flags are only updated when the `S` bit is set.

⟨`dpex()` *switch arith/logic opcode cases* 48a⟩+≡ (46d) ◁47a 54a▷
```
case OADD:
    ⟨dpex() if calltree, when add operation 106c⟩
    reg.r[rd] = o1 + o2;

    if(inst & Sbit) {
        if((XCAST(o1) + XCAST(o2)) & (1LL << 32))
            reg.cbit = true;
        else
            reg.cbit = false;
        reg.cc1 = o2;
        reg.cc2 = -o1;
        reg.compare_op = CCcmp;
    }
    return;
```
Uses `CCcmp` 64d, `OADD` 42, `Sbit` 47b, and `reg` 27d.

⟨*macro* `XCAST` 48b⟩≡ (128c)
```
#define XCAST(a) (uvlong)(ulong)a
```

5.4.7 `shift()`

A distinctive feature of the ARM ISA is the barrel shifter: one operand of any data-processing instruction can be shifted or rotated before the operation, at no extra cost. The shift type (`st`, 2 bits) selects logical left, logical right, arithmetic right, or rotate right; the shift count (`sc`) comes from either an immediate or a register. This is motivated by how common "multiply by a small power of two and add" is in real code: array indexing (`a[i]` becomes base + index × element size), pointer arithmetic, and address computation all follow this pattern. On x86 you would need a separate shift instruction followed by an add; on ARM, `ADD R1<<2, R2, R3` (R3 = R1 shifted left by 2 plus R2, i.e., index × 4 + base) is a single instruction.

⟨`itab` *elements* 48c⟩+≡ (27a) ◁45a 49a▷
```
// r<>#, r, r
[OAND +CARITH1] =  { Idp1,  "AND", Iarith },
[OEOR +CARITH1] =  { Idp1,  "EOR", Iarith },
[OSUB +CARITH1] =  { Idp1,  "SUB", Iarith },
[ORSB +CARITH1] =  { Idp1,  "RSB", Iarith },
[OADD +CARITH1] =  { Idp1,  "ADD", Iarith },
[OADC +CARITH1] =  { Idp1,  "ADC", Iarith },
[OSBC +CARITH1] =  { Idp1,  "SBC", Iarith },
[ORSC +CARITH1] =  { Idp1,  "RSC", Iarith },
[OTST +CARITH1] =  { Idp1,  "TST", Iarith },
[OTEQ +CARITH1] =  { Idp1,  "TEQ", Iarith },
[OCMP +CARITH1] =  { Idp1,  "CMP", Iarith },
[OCMN +CARITH1] =  { Idp1,  "CMN", Iarith },
[OORR +CARITH1] =  { Idp1,  "ORR", Iarith },
[OMOV +CARITH1] =  { Idp1,  "MOV", Iarith },
[OBIC +CARITH1] =  { Idp1,  "BIC", Iarith },
[OMVN +CARITH1] =  { Idp1,  "MVN", Iarith },
```
Uses `Iarith` 26b and `Idp1()` 48d.

⟨*function* `Idp1` 48d⟩≡ (128c)
```
/*
 * data processing instruction (R<>#),R,R
 */
void
```

```
Idp1(instruction inst)
{
    int rn, rd, rm, st, sc;
    long o1, o2;

    rn = (inst>>16) & 0xf;
    rd = (inst>>12) & 0xf;
    rm = inst & 0xf;
    st = (inst>>5) & 0x3;
    sc = (inst>>7) & 0x1f;

    o1 = reg.r[rn];
    ⟨adjust o1 if rn is REGPC 46a⟩
    o2 = reg.r[rm];
    ⟨adjust o2 if rm is REGPC 46b⟩

    o2 = shift(o2, st, sc, false);
    dpex(inst, o1, o2, rd);

    ⟨Idp1() trace ??⟩
    ⟨Idpx() compensate REGPC 46c⟩
}
```

Uses `dpex()` 46d, `reg` 27d, and `shift()` 50b.

⟨`itab` *elements* 49a⟩+≡ (27a) ◁48c 52c▷

```
// r<>r, r, r
[OAND +CARITH2] =  { Idp2,  "AND", Iarith },
[OEOR +CARITH2] =  { Idp2,  "EOR", Iarith },
[OSUB +CARITH2] =  { Idp2,  "SUB", Iarith },
[ORSB +CARITH2] =  { Idp2,  "RSB", Iarith },
[OADD +CARITH2] =  { Idp2,  "ADD", Iarith },
[OADC +CARITH2] =  { Idp2,  "ADC", Iarith },
[OSBC +CARITH2] =  { Idp2,  "SBC", Iarith },
[ORSC +CARITH2] =  { Idp2,  "RSC", Iarith },
[OTST +CARITH2] =  { Idp2,  "TST", Iarith },
[OTEQ +CARITH2] =  { Idp2,  "TEQ", Iarith },
[OCMP +CARITH2] =  { Idp2,  "CMP", Iarith },
[OCMN +CARITH2] =  { Idp2,  "CMN", Iarith },
[OORR +CARITH2] =  { Idp2,  "ORR", Iarith },
[OMOV +CARITH2] =  { Idp2,  "MOV", Iarith },
[OBIC +CARITH2] =  { Idp2,  "BIC", Iarith },
[OMVN +CARITH2] =  { Idp2,  "MVN", Iarith },
```

Uses `Iarith` 26b and `Idp2()` 49b.

⟨*function* `Idp2` 49b⟩≡ (128c)

```
/*
 * data processing instruction (R<>R),R,R
 */
void
Idp2(instruction inst)
{
    int rn, rd, rm, rs, st;
    long o1, o2, o3;

    rn = (inst>>16) & 0xf;
    rd = (inst>>12) & 0xf;
    rm = inst & 0xf;
    st = (inst>>5) & 0x3;
    rs = (inst>>8) & 0xf;
```

```
    o1 = reg.r[rn];
    ⟨adjust o1 if rn is REGPC 46a⟩
    o2 = reg.r[rm];
    ⟨adjust o2 if rm is REGPC 46b⟩
    o3 = reg.r[rs];
    ⟨adjust o3 if rs is REGPC 50a⟩

    o2 = shift(o2, st, o3, true);
    dpex(inst, o1, o2, rd);

    ⟨Idp2() trace ??⟩
    ⟨Idpx() compensate REGPC 46c⟩
}
```

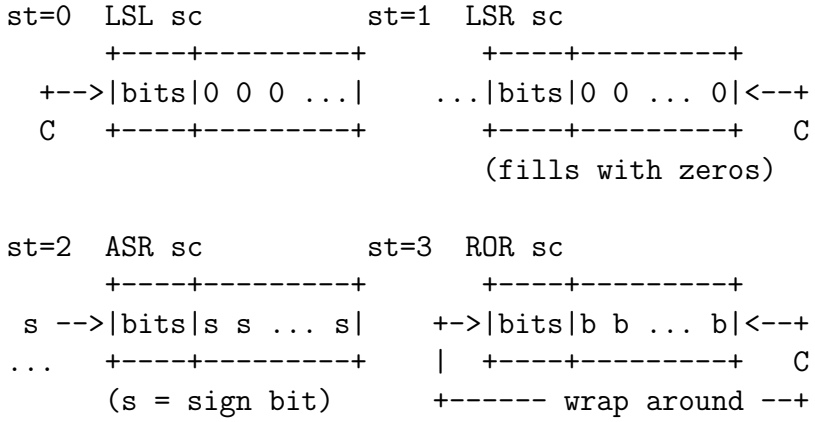
Uses dpex() 46d, reg 27d, and shift() 50b.

⟨*adjust o3 if rs is REGPC* 50a⟩≡ (49b)

```
if(rs == REGPC)
    o3 += 8;
```

Uses REGPC 28a.

The four shift modes correspond to the 2-bit `st` field (bits 5–6 of the instruction word). Here is what each one does to a 32-bit operand, with the bit that lands in `reg.cout` marked as `C`:



The shift count `sc` comes either from an immediate field (`Idp1`) or from a register (`Idp2`, which passes the low 8 bits of `R[rs]`). The special case `sc == 0` is handled separately because the ARM spec gives it idiosyncratic semantics per shift type: e.g., `LSR #0` actually means "shift by 32" so `cout` is the top bit, while `ROR #0` with an immediate means RRX (rotate right through the old carry). Real ARM cores implement this as a dedicated combinational barrel shifter on the B-bus so the shift is free in the instruction's cycle count; `5i` just uses a C switch.

⟨*function* shift 50b⟩≡ (128c)

```
long
shift(long v, int st, int sc, bool isreg)
{
    ⟨shift() if sc is 0 51⟩
    else {
        switch(st) {
        case 0: /* logical left */
            reg.cout = (v >> (32 - sc)) & 1;
            v = v << sc;
            break;
        case 1: /* logical right */
            reg.cout = (v >> (sc - 1)) & 1;
            v = (ulong)v >> sc;
            break;
```

```
        case 2: /* arith right */
            if(sc >= 32) {
                reg.cout = (v >> 31) & 1;
                if(reg.cout)
                    v = 0xFFFFFFFF;
                else
                    v = 0;
            }
            else {
                reg.cout = (v >> (sc - 1)) & 1;
                v = (long)v >> sc;
            }
            break;
        case 3: /* rotate right */
            reg.cout = (v >> (sc - 1)) & 1;
            v = (v << (32-sc))
                 |
                ((ulong)v >> sc);
            break;
        }
    }
    return v;
}
```

Uses reg 27d.

The distinction between logical right (case 1) and arithmetic right (case 2) is subtle but important. Both shift bits to the right, but they differ in what fills the vacated high bits. Logical right shift fills with zeros—it treats the value as unsigned, so shifting `0xFF000000` right by 8 gives `0x00FF0000`. Arithmetic right shift fills with copies of the sign bit —it preserves the sign of a two's complement number, so shifting `-256` (`0xFFFFFF00`) right by 8 gives `-1` (`0xFFFFFFFF`) instead of a large positive number. In C, this corresponds to the difference between `(ulong)v >> sc` (logical) and `(long)v >> sc` (arithmetic), which is exactly what the code does. There is no "arithmetic left" because left shift is the same for signed and unsigned values—zeros always fill the low bits.

⟨shift() *if sc is 0* 51⟩≡ (50b)

```
if(sc == 0) {
    switch(st) {
    case 0: /* logical left */
        reg.cout = reg.cbit;
        break;
    case 1: /* logical right */
        reg.cout = (v >> 31) & 1;
        break;
    case 2: /* arith right */
        reg.cout = reg.cbit;
        break;
    case 3: /* rotate right */
        if(isreg) {
            reg.cout = reg.cbit;
        }
        else {
            reg.cout = v & 1;
            v = ((ulong)v >> 1) | (reg.cbit << 31);
        }
    }
}
```

Uses reg 27d.

5.4.8 Immediate values

The fourth addressing mode embeds a small constant directly in the instruction word (class 1, i.e., bit 25 set). The immediate is encoded as an 8-bit value rotated right by twice a 4-bit field. The "twice" doubles the range of rotation positions: with 4 bits you can encode 0–15, but multiplying by 2 gives even rotations 0, 2, 4, . . . , 30, reaching any even bit position in the 32-bit word. There is no real loss from skipping odd rotations, because an 8-bit pattern at an odd position can usually be represented by adjusting the 8-bit value with an even rotation instead. This clever encoding can represent many common constants (powers of two, byte masks like `0xFF00`, page-aligned values) in just 12 bits:

⟨`arm_class()` *class cases* 52a⟩+≡ (41) ◁43b 55b▷
```
case 1: /* data processing i,r,r */
 op = CARITH3 + ((w >> 21) & 0xf);
 break;
```
Uses `CARITH3` 52b.

⟨*arith/logic opcodes* 52b⟩+≡ (26a) ◁44a 53a▷
```
CARITH3 = 48, // i,r,r
```

⟨`itab` *elements* 52c⟩+≡ (27a) ◁49a 53c▷
```
//i,r,r
[OAND +CARITH3] =  { Idp3,  "AND", Iarith },
[OEOR +CARITH3] =  { Idp3,  "EOR", Iarith },
[OSUB +CARITH3] =  { Idp3,  "SUB", Iarith },
[ORSB +CARITH3] =  { Idp3,  "RSB", Iarith },
[OADD +CARITH3] =  { Idp3,  "ADD", Iarith },
[OADC +CARITH3] =  { Idp3,  "ADC", Iarith },
[OSBC +CARITH3] =  { Idp3,  "SBC", Iarith },
[ORSC +CARITH3] =  { Idp3,  "RSC", Iarith },
[OTST +CARITH3] =  { Idp3,  "TST", Iarith },
[OTEQ +CARITH3] =  { Idp3,  "TEQ", Iarith },
[OCMP +CARITH3] =  { Idp3,  "CMP", Iarith },
[OCMN +CARITH3] =  { Idp3,  "CMN", Iarith },
[OORR +CARITH3] =  { Idp3,  "ORR", Iarith },
[OMOV +CARITH3] =  { Idp3,  "MOV", Iarith },
[OBIC +CARITH3] =  { Idp3,  "BIC", Iarith },
[OMVN +CARITH3] =  { Idp3,  "MVN", Iarith },
```
Uses `Iarith` 26b and `Idp3()` 52d.

⟨*function* `Idp3` 52d⟩≡ (128c)
```
/*
 * data processing instruction #<>#,R,R
 */
void
Idp3(instruction inst)
{
    int rn, rd, sc;
    long o1, o2;

    rn = (inst>>16) & 0xf;
    rd = (inst>>12) & 0xf;
    o1 = reg.r[rn];
    ⟨adjust o1 if rn is REGPC 46a⟩

    o2 = inst & 0xff;
    sc = (inst>>7) & 0x1e;
    o2 = (o2 >> sc) | (o2 << (32 - sc)); // rotate

    dpex(inst, o1, o2, rd);
```

⟨Idp3() *trace* ??⟩
⟨Idpx() *compensate REGPC* 46c⟩
}

Uses dpex() 46d and reg 27d.

5.4.9 64 bits target (signed/unsigned) multiplication

Multiplying two 32-bit numbers can produce a result that does not fit in 32 bits. In C, this happens whenever the compiler needs to support `long long` (64-bit) arithmetic on a 32-bit architecture, or even with plain `int` multiplication when the result is used in a wider context (e.g., `size_t a = b * c` where overflow must be detected). We already saw this problem in `5i` itself: the `XCAST` macro in Section 5.4.6 promotes 32-bit operands to 64 bits to detect carry on addition. The long multiply instructions (`MULLU`, `MULL`, etc.) handle this by storing the 64-bit result across two registers (`Rd` for the high half, `Rn` for the low half). The signed (`MULL`) and unsigned (`MULLU`) variants differ in how they extend the operands before multiplying.

⟨*arith/logic opcodes* 53a⟩+≡ (26a) ◁52b
```
OMULLU  = 66,
OMULALU = 67,
OMULL   = 68,
OMULAL  = 69,
```

⟨arm_class() *class 0, when x == 0x9, if bit 23 set* 53b⟩≡ (43d)
```
if(w & (1<<23)) { /* mullu */
    op = CMUL+2;
    if(w & (1<<22)) /* mull */
        op = CMUL+4;
}
```

Uses CMUL 43e.

⟨itab *elements* 53c⟩+≡ (27a) ◁52c 56a▷
```
[OMULLU]  =  { Imull, "MULLU", Iarith },
[OMULALU] =  { Imull, "MULALU", Iarith },
[OMULL]   =  { Imull, "MULL",  Iarith },
[OMULAL]  =  { Imull, "MULAL", Iarith },
```

Uses Iarith 26b and Imull() 53d.

⟨*function* Imull 53d⟩≡ (128c)
```
void
Imull(instruction inst)
{
    vlong v;
    int rs, rd, rm, rn;

    rd = (inst>>16) & 0xf;
    rn = (inst>>12) & 0xf;
    rs = (inst>>8) & 0xf;
    rm = inst & 0xf;

    if(rd == REGPC || rn == REGPC || rs == REGPC || rm == REGPC
      || rd == rm || rn == rm || rd == rn
      )
        undef(inst);

    if(inst & (1<<22)){ // mull
        v = (vlong)reg.r[rm] * (vlong)reg.r[rs];
        if(inst & (1 << 21)) // mull and accumulate
```

```
            v += reg.r[rn];
        }else{ // mullu
            v = XCAST(reg.r[rm]) * XCAST(reg.r[rs]);
            if(inst & (1 << 21)) // mullu and accumulate
                v += (ulong)reg.r[rn];
        }
        reg.r[rd] = v >> 32;
        reg.r[rn] = v;

        ⟨Imull() trace ??⟩
    }
```

Uses REGPC 28a, reg 27d, and undef() 122d.

5.4.10 Misc instructions

The remaining data-processing cases are less common. CMN (compare negative) is like CMP but adds instead of subtracting. RSB (reverse subtract) computes o2 - o1 instead of o1 - o2, useful when the immediate is the value you want to subtract from. ADC, SBC, and RSC (add/subtract with carry) are left unimplemented—the Plan 9 compiler 5c never generates them. Finally, MOV and MVN (move / move negated) set a register to a value or its bitwise complement; despite their names, they live in the arithmetic opcode space rather than in load/store.

⟨dpex() *switch arith/logic opcode cases* 54a⟩+≡ (46d) ◁48a 54b▷

```
    case OCMN:
        if(inst & Sbit) { // not always true?
            reg.cc1 = o1;
            reg.cc2 = -o2;
            reg.compare_op = CCcmp;
        }
        return;
```

Uses CCcmp 64d, OCMN 42, Sbit 47b, and reg 27d.

⟨dpex() *switch arith/logic opcode cases* 54b⟩+≡ (46d) ◁54a 54c▷

```
    case  ORSB:
        reg.r[rd] = o2 - o1;
        if(inst & Sbit) {
            reg.cc1 = o2;
            reg.cc2 = o1;
            reg.compare_op = CCcmp;
        }
        return;
```

Uses CCcmp 64d, ORSB 42, Sbit 47b, and reg 27d.

⟨dpex() *switch arith/logic opcode cases* 54c⟩+≡ (46d) ◁54b 54d▷

```
    case  OADC:
    case  OSBC:
    case  ORSC:
        undef(inst);
```

Uses OADC 42, ORSC 42, OSBC 42, and undef() 122d.

⟨dpex() *switch arith/logic opcode cases* 54d⟩+≡ (46d) ◁54c 63a▷

```
    case OMOV:
        reg.r[rd] = o2;
        cbit = true;
        break;
    case OMVN:
        reg.r[rd] = ~o2;
        cbit = true;
        break;
```

Uses OMOV 42, OMVN 42, and reg 27d.

5.5 Memory

The memory section handles load and store instructions. Like arithmetic, the opcode space is factored along two axes: the data size (word, byte, half-word, signed byte) and the addressing mode (immediate offset vs. register offset). The same `CMEM_BASIS + CMEMn` indexing scheme maps all variants into a contiguous region of `itab`.

5.5.1 Opcode extraction

⟨*memory opcodes* 55a⟩≡ (26a) 55d▷
```
OLDW    = 70,
OLDB    = 71,
OSTW    = 72,
OSTB    = 73,
```

⟨`arm_class()` *class cases* 55b⟩+≡ (41) ◁52a 55c▷
```
case 2: /* load/store byte/word i(r) */
 //                          OxxB?              OSTx?
 op = CMEM_BASIS + CMEM0 + ((w >> 22) & 0x1) + ((w >> 19) & 0x2);
 break;
```
Uses `CMEM0` 55d and `CMEM_BASIS` 55d.

⟨`arm_class()` *class cases* 55c⟩+≡ (41) ◁55b 60b▷
```
case 3: /* load/store byte/word (r)(r) */
 //                          OxxB?              OSTx?
 op = CMEM_BASIS + CMEM1 + ((w >> 22) & 0x1) + ((w >> 19) & 0x2);
 break;
```
Uses `CMEM1` 55d and `CMEM_BASIS` 55d.

⟨*memory opcodes* 55d⟩+≡ (26a) ◁55a 55e▷
```
CMEM_BASIS = 70,
CMEM0 = 0, // i(r)
CMEM1 = 4, // (r),(r)
CMEM2 = 8, // byte signed or half word
```

⟨*memory opcodes* 55e⟩+≡ (26a) ◁55d 55g▷
```
OLDH    = 78,
OLDBU   = 79,
OSTH    = 80,
OSTBU   = 81,
```

⟨`arm_class()` *class 0, if x has 0x9 bits* 55f⟩≡ (43b)
```
if((x & 0x9) == 0x9) {  /* ld/st byte/half s/u */
     //                          OxxBU?              OSTx?
     op = CMEM_BASIS + CMEM2 + ((w >> 22) & 0x1) + ((w >> 19) & 0x2);
     break;
}
```
Uses `CMEM2` 55d and `CMEM_BASIS` 55d.

⟨*memory opcodes* 55g⟩+≡ (26a) ◁55e 60a▷
```
OSWPW = 82,
OSWPBU = 83,
```

⟨`arm_class()` *class 0, when x == 0x9, if bit 24 set* 55h⟩≡ (43d)
```
if(w & (1<<24)) {
    op = OSWPW;
    if(w & (1<<22))
         op = OSWPBU;
    break;
}
```
Uses `OSWPBU` 55g and `OSWPW` 55g.

5.5.2 Memory swap

As with multiplication in the arithmetic section, I present `SWP` before the general load/store because it is simpler: it just swaps a register value with a memory location in a single atomic operation. On real hardware, atomicity matters for implementing `_tas()` (test-and-set) on multiprocessor systems; in `5i`'s single-processor emulation, the atomicity is trivially guaranteed. The `bbit` (bit 22) selects between word and byte variants. The byte variant is called `SWPBU` (not `SWPB`) because loading a byte into a 32-bit register always zero-extends—the "U" stands for unsigned. In load/store, there is a distinction between `MOVB` (sign-extend, for `signed char`) and `MOVBU` (zero-extend, for `unsigned char`), but for swap only the unsigned variant exists since sign extension makes no sense for an atomic exchange.

⟨`itab` *elements* 56a⟩+≡ (27a) ◁53c 56c▷
```
[OSWPW] =   { Iswap,  "SWPW", Imem },
[OSWPBU] =  { Iswap,  "SWPBU",Imem },
```
Uses `Imem` 26b and `Iswap()` 56b.

⟨*function* `Iswap` 56b⟩≡ (128c)
```
void
Iswap(instruction inst)
{
    int rn, rd, rm;
    ulong address, value;
    bool bbit;

    bbit = inst & (1<<22); // BU?

    rn = (inst>>16) & 0xf;
    rd = (inst>>12) & 0xf;
    rm = (inst>>0) & 0xf;

    address = reg.r[rn];
    if(bbit) {
        value = getmem_b(address);
        putmem_b(address, reg.r[rm]);
    } else {
        value = getmem_w(address);
        putmem_w(address, reg.r[rm]);
    }
    reg.r[rd] = value;

    ⟨`Iswap()` *trace* ??⟩
}
```
Uses `getmem_b()` 72a, `getmem_w()` 72c, `putmem_b()` 73d, `putmem_w()` 74b, and `reg` 27d.

The `getmem_b`[72a]/`getmem_w`[72c] and `putmem_b`[73d]/`putmem_w`[74b] functions translate virtual addresses to the simulated memory and perform the actual read or write. They are described in Chapter 6.

5.5.3 Load/store

With `Iswap` as a warm-up, we are now ready for the general load/store handler `Imem1`[57b], which is the most complex instruction handler in `5i`. It must handle all combinations of load vs. store, word vs. byte, immediate vs. register offset, pre- vs. post-indexing, and optional write-back—all controlled by individual bits in the instruction word:

⟨`itab` *elements* 56c⟩+≡ (27a) ◁56a 57a▷
```
// load/store w/ub i,r
[OLDW +CMEMO] =  { Imem1,  "MOVW", Imem },
[OLDB +CMEMO] =  { Imem1,  "MOVB", Imem },
```

```
[OSTW +CMEM0] =  { Imem1,  "MOVW", Imem },
[OSTB +CMEM0] =  { Imem1,  "MOVB", Imem },
```

Uses Imem 26b and Imem1() 57b.

⟨itab *elements* 57a⟩+≡ (27a) ◁56c 58▷

```
// load/store r,r
[OLDW +CMEM1] =  { Imem1,  "MOVW", Imem },
[OLDB +CMEM1] =  { Imem1,  "MOVB", Imem },
[OSTW +CMEM1] =  { Imem1,  "MOVW", Imem },
[OSTB +CMEM1] =  { Imem1,  "MOVB", Imem },
```

Uses Imem 26b and Imem1() 57b.

The function first extracts six single-bit flags from the instruction word, each controlling one axis of variation:

- `bit25`: offset is a (shifted) register or an immediate?
- `prebit`: add the offset before (pre-indexing) or after (post-indexing) the memory access?
- `ubit`: offset is positive (up) or negative (down)?
- `bbit`: transfer a byte or a word?
- `wbit`: write the computed address back into `Rn`?
- `lbit`: load (LDR) or store (STR)?

The pre/post-indexing and write-back modes are how ARM implements stack push/pop and auto-increment addressing in a single instruction. For example, `MOVW.P \#4(R13), R15` (see the ASSEMBLER book [Pad15a]) uses post-indexing: it loads from `R13`, then increments `R13` by 4—effectively a "pop" in one instruction.

⟨*function* Imem1 57b⟩≡ (128c)

```
/*
 * load/store word/byte
 */
void
Imem1(instruction inst)
{
    int rn, rd, off, rm, sc, st;
    ulong address, value;
    bool prebit, ubit, bbit, wbit, lbit, bit25;

    bit25 = inst & (1<<25); // rm or I?
    prebit = inst & (1<<24); // Pre indexing?
    ubit = inst & (1<<23); // Up offset?
    bbit = inst & (1<<22); // Byte or Word?
    wbit = inst & (1<<21); // Write back address in rn?
    lbit = inst & (1<<20); // LDR or STR?

    rn = (inst>>16) & 0xf;
    rd = (inst>>12) & 0xf;

    SET(st);
    SET(sc);
    SET(rm);

    if(bit25) {
        // rm<>I(...)
        rm = inst & 0xf;
        st = (inst>>5) & 0x3;
        sc = (inst>>7) & 0x1f;
```

```
            off = reg.r[rm];
            if(rm == REGPC)
                off += 8;
            off = shift(off, st, sc, false);
        } else {
            // I(...)
            off = inst & 0xfff;
        }

        if(!ubit)
            off = -off;
        if(rn == REGPC)
            off += 8;

        address = reg.r[rn];
        if(prebit)
            address += off;

        if(lbit) {
            // LDR
            if(bbit)
                value = getmem_b(address);
            else
                value = getmem_w(address);
            if(rd == REGPC)
                value -= 4;
            reg.r[rd] = value;
        } else {
            // STR
            value = reg.r[rd];
            if(rd == REGPC)
                value -= 4;
            if(bbit)
                putmem_b(address, value);
            else
                putmem_w(address, value);
        }
        if(!prebit || wbit)
            reg.r[rn] += off;

        ⟨Imem1() trace ??⟩
    }
```

Uses REGPC 28a, getmem_b() 72a, getmem_w() 72c, putmem_b() 73d, putmem_w() 74b, reg 27d, and shift() 50b.

The flow is: compute the offset (from an immediate or a shifted register), apply the sign (`ubit`), compute the effective address (adding the offset before or after the access depending on `prebit`), perform the load or store (`lbit`), and finally write back the updated address to `Rn` if post-indexing or if `wbit` is set. The last line (`if(!prebit || wbit)`) captures both post-indexing (which always writes back) and pre-indexing with explicit write-back—the original code used a double negation (`if(!(prebit \&\& !wbit))`) which is equivalent but harder to read.

⟨itab *elements* 58⟩+≡ (27a) ◁57a 60d▷

```
    // load/store h/sb
    [OLDH]  =  { Imem2,  "MOV", Imem },
    [OLDBU] =  { Imem2,  "MOV", Imem },
    [OSTH]  =  { Imem2,  "MOV", Imem },
    [OSTBU] =  { Imem2,  "MOV", Imem },
```

Uses Imem 26b and Imem2() 59.

⟨*function* Imem2 59⟩≡ (128c)

```
/*
 * load/store unsigned byte/half word
 */
void
Imem2(instruction inst)
{
    int rn, rd, off, rm;
    ulong address, value;
    bool prebit, ubit, hbit, sbit, wbit, lbit, bit22;

    prebit = inst & (1<<24); // Pre indexing?
    ubit = inst & (1<<23); // Up offset?
    bit22 = inst & (1<<22);
    wbit = inst & (1<<21); // Write back address in rn
    lbit = inst & (1<<20); // LDR or STR?

    sbit = inst & (1<<6); // Signed?
    hbit = inst & (1<<5); // Half word or byte?

    rn = (inst>>16) & 0xf;
    rd = (inst>>12) & 0xf;

    SET(rm);
    if(bit22) {
        // I(...)
        off = ((inst>>4) & 0xf0) | (inst & 0xf);
    } else {
        // rm(...)
        rm = inst & 0xf;
        off = reg.r[rm];
        if(rm == REGPC)
            off += 8;
    }
    if(!ubit)
        off = -off;
    if(rn == REGPC)
        off += 8;

    address = reg.r[rn];
    if(prebit)
        address += off;

    if(lbit) {
        // LDR
        if(hbit) {
            value = getmem_h(address);
            if(sbit && (value & 0x8000))
                value |= 0xffff0000;
        } else {
            value = getmem_b(address);
            if(value & 0x80)
                value |= 0xffffff00;
        }
        if(rd == REGPC)
            value -= 4;
        reg.r[rd] = value;
    } else {
        // STR
        value = reg.r[rd];
```

```
        if(rd == REGPC)
            value -= 4;
        if(hbit) {
            putmem_h(address, value);
        } else {
            putmem_b(address, value);
        }
    }
    if(!prebit || wbit)
        reg.r[rn] += off;

    ⟨Imem2() trace ??⟩
}
```

Uses REGPC 28a, getmem_b() 72a, getmem_h() 72b, putmem_b() 73d, putmem_h() 74a, and reg 27d.

5.5.4 Multi registers load/store

LDM/STM (load/store multiple, called MOVM in Plan 9 syntax) transfer a set of registers to/from memory in a single instruction, selected by a 16-bit bitmask. The Plan 9 compiler 5c generates MOVM for function prologues and epilogues: in the Plan 9 calling convention, the callee is responsible for saving any registers that are in use by the caller and that it will clobber. STM pushes these registers onto the stack at entry, LDM restores them at return (see the COMPILER book [Pad16a] and ASSEMBLER book [Pad15a]). For example, a function that uses R1 through R4 would begin with MOVM.DB.W [R1-R4], (R13) (push R1–R4 onto the stack, decrementing R13) and end with MOVM.IA.W (R13), [R1-R4] (pop them back, incrementing R13). The suffixes control the direction and timing: .D/.I = decrement/increment (ubit), .B/.A = before/after the transfer (prebit), and .W = write back the final address into Rn (wbit). So .DB means "decrement before each store" (stack grows down), and .IA means "increment after each load" (stack grows back up).

⟨*memory opcodes* 60a⟩+≡ (26a) ◁55g 60c▷
```
OLDM = 84,
OSTM = 85,
```

⟨arm_class() *class cases* 60b⟩+≡ (41) ◁55c 62b▷
```
case 4: /* block data transfer (r)(r) */
 op = CBLOC + ((w >> 20) & 0x1);
 break;
```

Uses CBLOC 60c.

⟨*memory opcodes* 60c⟩+≡ (26a) ◁60a
```
CBLOC   = 84,
```

⟨itab *elements* 60d⟩+≡ (27a) ◁58 62d▷
```
// block move r,r
[OLDM] =  { Ilsm,  "LDM", Imem },
[OSTM] =  { Ilsm,  "STM", Imem },
```

Uses Ilsm() 60e and Imem 26b.

⟨*function* Ilsm 60e⟩≡ (128c)
```
void
Ilsm(instruction inst)
{
    bool prebit, ubit, sbit, wbit, lbit;
    int i, rn, reglist;
    ulong address, predelta, postdelta;

    prebit = (inst>>24) & 0x1;
    ubit = (inst>>23) & 0x1;
```

```
    sbit = (inst>>22) & 0x1;
    wbit = (inst>>21) & 0x1;
    lbit = (inst>>20) & 0x1;
    rn =   (inst>>16) & 0xf;
    reglist = inst & 0xffff;

    if(reglist & 0x8000)
        undef(reg.instr);
    if(sbit)
        undef(reg.instr);

    address = reg.r[rn];

    if(prebit) {
        predelta = 4;
        postdelta = 0;
    } else {
        predelta = 0;
        postdelta = 4;
    }
    if(ubit) {
        for (i = 0; i < 16; ++i) {
            if(!(reglist & (1 << i)))
                continue;
            address += predelta;
            if(lbit)
                reg.r[i] = getmem_w(address);
            else
                putmem_w(address, reg.r[i]);
            address += postdelta;
        }
    } else {
        for (i = 15; 0 <= i; --i) {
            if(!(reglist & (1 << i)))
                continue;
            address -= predelta;
            if(lbit)
                reg.r[i] = getmem_w(address);
            else
                putmem_w(address, reg.r[i]);
            address -= postdelta;
        }
    }
    if(wbit) {
        reg.r[rn] = address;
    }

    ⟨Ilsm() trace ??⟩
}
```

Uses getmem_w() 72c, putmem_w() 74b, reg 27d, and undef() 122d.

5.6 Control flow

ARM has only two branch instructions: B (branch) and BL (branch and link, i.e., function call). There is no separate "conditional branch" like x86's JNE/JEQ—any branch (or any instruction) can be made conditional via the condition code field (Section 5.6.5), so a conditional branch is simply B.EQ, B.GT, etc.

5.6.1 Opcode extraction

⟨*branching opcodes* 62a⟩≡ (26a) 62c▷
```
OB = 86,
OBL = 87,
```

⟨`arm_class()` *class cases* 62b⟩+≡ (41) ◁60b 66b▷
```
case 5: /* branch / branch link */
 op = CBRANCH + ((w >> 24) & 0x1);
 break;
```
Uses `CBRANCH` 62c.

⟨*branching opcodes* 62c⟩+≡ (26a) ◁62a
```
CBRANCH = 86,
```

5.6.2 Simple branching

⟨`itab` *elements* 62d⟩+≡ (27a) ◁60d 62f▷
```
// branch
[OB]  =  { Ib, "B", Ibranch },
```
Uses `Ib()` 62e and `Ibranch` 26b.

The branch offset is a 24-bit signed value, shifted left by 2 (since instructions are word-aligned), giving a range of ±32MB. The `+8` accounts for the ARM pipeline: the PC is two instructions ahead when the branch executes. The `-4` compensates for the `REGPC += 4` at the end of `run()`[40c]:

⟨*function* `Ib` 62e⟩≡ (128c)
```
void
Ib(instruction inst)
{
    long v;

    v = inst & 0xffffff; // 24 bits
    v = reg.r[REGPC] + (v << 2) + 8;
    ⟨Ib() trace ??⟩
    reg.r[REGPC] = v - 4;
}
```
Uses `REGPC` 28a and `reg` 27d.

5.6.3 Branch and link

`BL` (branch and link) is the function call instruction: it saves the return address in the link register (`R14`) before branching. The callee must save `R14` to the stack if it makes further calls; leaf functions (see the Linker book [Pad15b] for how the linker optimizes them) can return with a simple `MOV R14, R15`:

⟨`itab` *elements* 62f⟩+≡ (27a) ◁62d 66c▷
```
[OBL] =  { Ibl, "BL", Ibranch },
```
Uses `Ibl()` 62g and `Ibranch` 26b.

⟨*function* `Ibl` 62g⟩≡ (128c)
```
void
Ibl(instruction inst)
{
    long v;
    Symbol s;

    v = inst & 0xffffff;
    v = reg.r[REGPC] + (v << 2) + 8;
```

```
    ⟨Ibl() trace ??⟩
    ⟨Ibl() if calltree 107a⟩
    reg.r[REGLINK] = reg.r[REGPC] + 4;
    reg.r[REGPC] = v - 4;
}
```

Uses REGLINK 28a, REGPC 28a, and reg 27d.

5.6.4 Comparisons

Comparisons are the bridge between arithmetic and conditional execution: they set the flags that subsequent conditionally-executed instructions will test. CMP is actually SUB with the S bit always set and the result discarded—it only updates cc1 and cc2. This is why OSUB falls through to OCMP in the code below: both compute the same flags, but SUB also stores the result in Rd. Similarly, TST is AND with the result discarded, and TEQ is EOR with the result discarded.

⟨dpex() *switch arith/logic opcode cases* 63a⟩+≡ (46d) ◁54d 63b▷

```
case OSUB:
    reg.r[rd] = o1 - o2;
    // Fallthrough
case OCMP:
    if(inst & Sbit) {
        reg.cc1 = o1;
        reg.cc2 = o2;
        reg.compare_op = CCcmp;
    }
    return;
```

Uses CCcmp 64d, OCMP 42, OSUB 42, Sbit 47b, and reg 27d.

⟨dpex() *switch arith/logic opcode cases* 63b⟩+≡ (46d) ◁63a

```
case  OTST:
    if(inst & Sbit) {
        reg.cc1 = o1;
        reg.cc2 = o2;
        reg.compare_op = CCtst;
    }
    return;
case  OTEQ:
    if(inst & Sbit) { // not always true?
        reg.cc1 = o1;
        reg.cc2 = o2;
        reg.compare_op = CCteq;
    }
    return;
```

Uses CCteq 64d, CCtst 64d, OTEQ 42, OTST 42, Sbit 47b, and reg 27d.

5.6.5 ARM Conditional execution

ARM's most distinctive feature is conditional execution: every instruction has a 4-bit condition code in bits 28–31. Before executing, the processor checks this condition against the current flags (set by a previous CMP, TST, or any S-suffixed instruction). If the condition is false, the instruction is skipped. This avoids short branches for simple if/else sequences, which was important on early ARM cores without branch prediction. The implementation in 5i is unusual: instead of maintaining N/Z/C/V flags like real hardware, it stores the two comparison operands (cc1, cc2) and the comparison type (compare_op), then re-evaluates the condition on

each instruction. This is slower but simpler than tracking individual flags. The condition `0xe` ("always") means unconditional execution—most instructions use this.

⟨Registers *other fields* 64a⟩+≡ (27c) ◁47d 64c▷
```
// actually only 4 bits (16 possibilities)
int instr_cond;
```

⟨run() *set reg.cond* 64b⟩≡ (40c)
```
reg.instr_cond = (reg.instr>>28) & 0xf;
```
Uses reg 27d.

⟨Registers *other fields* 64c⟩+≡ (27c) ◁64a 64f▷
```
// enum<compare_op>
int compare_op;
```

⟨*enum* compare_op 64d⟩≡ (124)
```
enum compare_op
{
    CCcmp,
    CCtst,
    CCteq,
};
```

⟨run() *switch* reg.compare_op *to set* execute 64e⟩≡ (40c)
```
switch(reg.compare_op) {
case CCcmp:
    execute = runcmp(); // use reg.instr_cond
    break;
case CCteq:
    execute = runteq();
    break;
case CCtst:
    execute = runtst();
    break;
default:
    Bprint(bout, "unimplemented compare operation %x\n",
        reg.compare_op);
    return;
}
```
Uses CCcmp 64d, CCteq 64d, CCtst 64d, bout 31d, reg 27d, runcmp() 64g, runteq() 65b, and runtst() 65c.

⟨Registers *other fields* 64f⟩+≡ (27c) ◁64c
```
long cc1;
long cc2;
```

⟨*function* runcmp 64g⟩≡ (128c)
```
bool
runcmp(void)
{
    switch(reg.instr_cond) {
    case 0x0: /* eq */ return (reg.cc1 == reg.cc2);
    case 0x1: /* ne */ return (reg.cc1 != reg.cc2);
    case 0x2: /* hs */ return ((ulong)reg.cc1 >= (ulong)reg.cc2);
    case 0x3: /* lo */ return ((ulong)reg.cc1 < (ulong)reg.cc2);
    case 0x4: /* mi */ return (reg.cc1 - reg.cc2 < 0);
    case 0x5: /* pl */ return (reg.cc1 - reg.cc2 >= 0);
    case 0x8: /* hi */ return ((ulong)reg.cc1 > (ulong)reg.cc2);
    case 0x9: /* ls */ return ((ulong)reg.cc1 <= (ulong)reg.cc2);
    case 0xa: /* ge */ return (reg.cc1 >= reg.cc2);
    case 0xb: /* lt */ return (reg.cc1 < reg.cc2);
```

```
    case 0xc: /* gt */ return (reg.cc1 > reg.cc2);
    case 0xd: /* le */ return (reg.cc1 <= reg.cc2);

    case 0xe: /* al */ return true;
    case 0xf: /* nv */ return false;
    default:
        Bprint(bout, "unimplemented condition prefix %x (%ld %ld)\n",
            reg.instr_cond, reg.cc1, reg.cc2);
        undef(reg.instr);
        return false;
    }
}
```

Uses bout 31d, reg 27d, and undef() 122d.

⟨*constant* SIGNBIT 65a⟩≡ (33e)

```
SIGNBIT  = 0x80000000,
```

⟨*function* runteq 65b⟩≡ (128c)

```
bool
runteq(void)
{
    long res = reg.cc1 ^ reg.cc2;

    switch(reg.instr_cond) {
    case 0x0: /* eq */ return res == 0;
    case 0x1: /* ne */ return res != 0;
    case 0x4: /* mi */ return (res & SIGNBIT) != 0;
    case 0x5: /* pl */ return (res & SIGNBIT) == 0;
    case 0xe: /* al */ return true;
    case 0xf: /* nv */ return false;
    default:
        Bprint(bout, "unimplemented condition prefix %x (%ld %ld)\n",
            reg.instr_cond, reg.cc1, reg.cc2);
        undef(reg.instr);
        return false;
    }
}
```

Uses SIGNBIT 65a, bout 31d, reg 27d, and undef() 122d.

⟨*function* runtst 65c⟩≡ (128c)

```
bool
runtst(void)
{
    long res = reg.cc1 & reg.cc2;

    switch(reg.instr_cond) {
    case 0x0: /* eq */ return res == 0;
    case 0x1: /* ne */ return res != 0;
    case 0x4: /* mi */ return (res & SIGNBIT) != 0;
    case 0x5: /* pl */ return (res & SIGNBIT) == 0;
    case 0xe: /* al */ return true;
    case 0xf: /* nv */ return false;
    default:
        Bprint(bout, "unimplemented condition prefix %x (%ld %ld)\n",
            reg.instr_cond, reg.cc1, reg.cc2);
        undef(reg.instr);
        return false;
    }
}
```

Uses SIGNBIT 65a, bout 31d, reg 27d, and undef() 122d.

5.7 Software interrupt

The `SWI` (software interrupt) instruction is how user programs make system calls. In a real ARM, it would trap to the kernel; in `5i`, it dispatches to `Ssyscall()`[77a] which emulates the system call by proxying it to the host OS (see Chapter 7).

⟨*syscall opcodes* 66a⟩≡ (26a)
```
OSWI = 88,
```

⟨`arm_class()` *class cases* 66b⟩+≡ (41) ◁62b
```
case 7: /* coprocessor crap */ // and syscall
 if((w >> 25) & 0x1)
   op = OSWI;
 else
   op = OUNDEF; // coprocessor stuff not handled
 break;
```
Uses `OSWI` 66a and `OUNDEF` 26a.

⟨`itab` *elements* 66c⟩+≡ (27a) ◁62f
```
[OSWI] =  { Ssyscall,  "SWI", Isyscall },
```
Uses `Isyscall` 26b and `Ssyscall()` 77a.

5.8 Unimplemented instructions

A user-mode emulator can cut a lot of corners that a system-mode emulator cannot. `5i` does not implement the coprocessor instructions (`MCR`/`MRC`), the exception return (`RFE`), the cache management ops, or any of the other privileged instructions that only the kernel would use. If the guest program executes one of these, the decode table routes it to `undef()`[122d], which prints a diagnostic and `longjmp`s back to the REPL via `errjmp`[122b]. This is the same philosophy Linux's `qemu-user` uses: trapping an unimplemented instruction and falling back to an error is much cheaper than implementing the full privileged mode, and no well-behaved user program should ever hit one.

Chapter 6

Memory

The previous chapter showed how the interpreter decodes and executes each ARM instruction. Many of those instructions—loads, stores, instruction fetches—need to access memory. In a real ARM processor, the MMU translates virtual addresses to physical addresses and handles page faults. In `5i`, the emulator proxies memory itself: virtual addresses from the emulated program map to host memory obtained via `malloc()` (stored in `Segment.table`, introduced in Chapter ??).

In this chapter I will first present `page_of_vaddr()`, the central function that translates a virtual address to a host pointer, including demand-loading pages on first access (page faults). I will then describe the TLB simulation used for profiling, the instruction fetch path `ifetch()` with its cache, and finally the `getmem` and `putmem` families that the interpreter calls for load and store instructions.

6.1 page_of_vaddr()

`page_of_vaddr()` is the memory system's core function. Given a virtual address, it walks the segment array looking for the segment that contains that address, then indexes into the segment's page table (`s->table`) to find the corresponding host pointer. If the page has already been allocated (the pointer is non-nil), it returns immediately. Otherwise, this is a page fault: the emulator allocates a new page and fills it according to the segment type—reading from the executable for Text and Data, or zero-filling for Bss and Stack.

⟨*function* `page_of_vaddr` 67⟩≡ (128a)

```
void*
page_of_vaddr(uintptr addr)
{
    Segment *s, *es;
    int off, foff, l, n;
    byte **p, *a;

    ⟨page_of_vaddr() TLB handling 69f⟩

    es = &memory.seg[Nseg];
    for(s = memory.seg; s < es; s++) {
        if(addr >= s->base && addr < s->end) {
            s->refs++;
            off = (addr - s->base)/BY2PG;
            p = &s->table[off];

            if(*p)
                return *p;

            // else page fault! no allocated memory there yet
            s->rss++;
```

```
            switch(s->type) {
            ⟨page_of_vaddr() page fault, switch segment type cases 68a⟩
            default:
                fatal(false, "page_of_vaddr");
            }
        }
    }
    // reach here if didn't find any segment with relevant range
    Bprint(bout, "User TLB miss vaddr 0x%.8lux\n", addr);
    Bflush(bout);
    longjmp(errjmp, 0);
    return nil;  /*to stop compiler whining*/
}
```

Uses BY2PG 33e, Nseg 28b, bout 31d, errjmp 122b, fatal() 122a, and memory 28d.

6.2 Page faults

The page fault handler implements demand loading: pages are only read from the executable file when first accessed, not all at once at startup. This mirrors how a real OS kernel handles page faults.

For the Text segment, the handler allocates a fresh page with `emalloc()` and reads the corresponding page from the executable file. The file offset is computed from the segment's `fileoff` base plus the page index times BY2PG.

⟨page_of_vaddr() *page fault, switch segment type cases 68a*⟩≡ (67) 68b▷

```
case Text:
    *p = emalloc(BY2PG);
    if(seek(text, s->fileoff+(off*BY2PG), 0) < 0)
        fatal(true, "page_of_vaddr text seek");
    if(read(text, *p, BY2PG) < 0)
        fatal(true, "page_of_vaddr text read");
    return *p;
```

Uses BY2PG 33e, Text 28b, emalloc() 123a, fatal() 122a, and text 31b.

The Data segment is similar but has an extra subtlety: the last page of initialised data may extend past `s->fileend`, into the region that should be zero-filled. The code reads what it can from the file, then `memset`s the remainder of the page to zero.

⟨page_of_vaddr() *page fault, switch segment type cases 68b*⟩+≡ (67) ◁68a 69a▷

```
case Data:
    *p = emalloc(BY2PG);
    foff = s->fileoff+(off*BY2PG);
    if(seek(text, foff, 0) < 0)
        fatal(true, "page_of_vaddr text seek");
    n = read(text, *p, BY2PG);
    if(n < 0)
        fatal(true, "page_of_vaddr text read");
    if(foff + n > s->fileend) {
        l = BY2PG - (s->fileend-foff);
        a = *p+(s->fileend-foff);
        memset(a, 0, l);
    }
    return *p;
```

Uses BY2PG 33e, Data 28b, emalloc() 123a, fatal() 122a, and text 31b.

Bss and Stack pages need no file I/O—they are simply allocated and implicitly zero-filled (since `emalloc()` calls `memset(, 0, )`).

⟨`page_of_vaddr()` *page fault, switch segment type cases* 69a⟩+≡ (67) ◁68b
```
case Bss:
case Stack:
    *p = emalloc(BY2PG);
    return *p;
```
Uses `BY2PG` 33e, `Bss` 28b, `Stack` 28b, and `emalloc()` 123a.

6.3 `Tlb`

Since `5i` proxies memory via host `malloc()`, it does not actually need a TLB for address translation. The TLB here is a simulation: it tracks which pages the emulated program accesses and records hit/miss statistics. This turns `5i` into a cache/TLB profiler—one can experiment with different TLB sizes and observe miss rates without modifying any hardware.

⟨*struct* `Tlb` 69b⟩≡ (124)
```
struct Tlb
{
    bool on;   /* Being updated */
    int  tlbsize;  /* Number of entries */
    // all pointers in array are at page granularity
    uintptr tlbent[Nmaxtlb]; /* Virtual address tags */

    int hit;   /* Number of successful tag matches */
    int miss;   /* Number of failed tag matches */
};
```
Uses `Nmaxtlb` 69c.

⟨*constant* `Nmaxtlb` 69c⟩≡ (124)
```
#define Nmaxtlb 64
```

⟨*global* `tlb` 69d⟩≡ (126b)
```
Tlb tlb;
```

⟨`main()` *tlb initialisation* 69e⟩≡ (31e)
```
tlb.on = true;
tlb.tlbsize = 24;
```
Uses `tlb` 69d.

⟨`page_of_vaddr()` *TLB handling* 69f⟩≡ (67)
```
if(tlb.on)
    dotlb(addr);
```
Uses `dotlb()` 69g and `tlb` 69d.

`dotlb()` simulates a TLB lookup. It masks the address down to page granularity, then searches the TLB array for a match. On a miss, it evicts a random entry—a simple but reasonable approximation of real TLB replacement policies.

⟨*function* `dotlb` 69g⟩≡ (128a)
```
void
dotlb(uintptr vaddr)
{
    ulong *l, *e;

    vaddr &= ~(BY2PG-1);
```

```
    e = &tlb.tlbent[tlb.tlbsize];
    for(l = tlb.tlbent; l < e; l++)
        if(*l == vaddr) {
            tlb.hit++;
            return;
        }

    tlb.miss++;
    tlb.tlbent[lnrand(tlb.tlbsize)] = vaddr;
}
```

Uses BY2PG 33e and tlb 69d.

6.4 ifetch()

ifetch() fetches a single ARM instruction from the emulated program's text segment. It first checks alignment (ARM instructions must be word-aligned), then updates the profiling counter and instruction cache, looks up the page via page_of_vaddr(), and finally reassembles the 32-bit instruction from four bytes in little-endian order.

The little-endian reassembly on the last line is worth unpacking. On disk and in the text segment, an instruction word is stored as four consecutive bytes, lowest-significance first:

```
memory layout at va = reg.ar  (little-endian)

 va+0    va+1    va+2    va+3
+-------+-------+-------+-------+
| 0x03  | 0x10  | 0x82  | 0xE0  |     bytes on the page
+---+---+---+---+---+---+---+---+
    |       |       |       |
    v       v<<8    v<<16   v<<24
    |       |       |       |
    +-------+-------+-------+
            |
            v
      0xE0821003      = reg.instr
      AL ADD R2,R1,R3 (ARM encoding of ADD R2,R3,R1 in Plan 9 syntax)
```

The code does not cast (uint*)va and dereference, which would be tempting but unsafe: the host might be big-endian, or have stricter alignment than the guest's 4-byte rule (e.g., old SPARC). Doing the byte-by-byte OR keeps 5i portable across host architectures and makes the endianness explicit in the source—a rare case where the slow path is also the correct path.

⟨*function* ifetch 70⟩≡ (128a)

```
instruction
ifetch(uintptr addr)
{
    byte *va;

    if(addr&3) {
        Bprint(bout, "Address error (I-fetch) vaddr %.8lux\n", addr);
        longjmp(errjmp, 0);
    }

    ⟨ifetch() instruction cache handling 71c⟩
    iprof[(addr-textbase)/PROFGRAN]++;
```

```
    va = page_of_vaddr(addr); // get page
    va += addr&(BY2PG-1); // restore offset in page

    return va[3]<<24 | va[2]<<16 | va[1]<<8 | va[0];
}
```

Uses BY2PG 33e, PROFGRAN 112b, bout 31d, errjmp 122b, iprof 112a, page_of_vaddr() 67, and textbase 34d.

6.5 The instruction cache

Like the TLB, the instruction cache is a simulation for profiling purposes—5i does not need a cache to function. The Icache structure holds a configurable line size, a tag array, and a pluggable hash function. Note however that updateicache() is currently a stub (USED(addr)), so the icache infrastructure is present but inactive for ARM.

⟨*struct* Icache 71a⟩≡ (124)

```
struct Icache
{
    bool on;   /* Turned on */

    int  linesize;  /* Line size in bytes */
    int  stall;   /* Cache stalls */
    int* lines;   /* Tag array */
    int* (*hash)(ulong); /* Hash function */
    char* hashtext;  /* What the function looks like */
};
```

⟨*global* icache 71b⟩≡ (126b)

```
Icache icache;
```

⟨ifetch() *instruction cache handling* 71c⟩≡ (70)

```
if(icache.on)
    updateicache(addr);
```

Uses icache 71b and updateicache() 71d.

⟨*function* updateicache 71d⟩≡ (126a)

```
void
updateicache(uintptr addr)
{
    USED(addr);
}
```

6.6 getmem_xxx()

The getmem family provides typed memory reads at byte (_b), half-word (_h), word (_w), and double-word (_v) granularity. These are the functions called by the load instructions in Chapter 5. Each function translates the virtual address to a host pointer via page_of_vaddr(), then reassembles the value from individual bytes in little-endian order.

For multi-byte reads, alignment matters: if the address is properly aligned, the bytes are guaranteed to be within a single page, so a single page_of_vaddr() call suffices. For unaligned accesses (getmem_h and getmem_w),

the code first reads the aligned word, then rotates the bytes to produce the correct result—a clever trick that avoids dealing with cross-page boundaries.

⟨*function* `getmem_b` 72a⟩≡ (128a)
```
byte
getmem_b(uintptr addr)
{
    byte *va;

    ⟨getmem_x() if membpt 110d⟩

    va = page_of_vaddr(addr);
    va += addr&(BY2PG-1);
    return va[0];
}
```
Uses `BY2PG` 33e and `page_of_vaddr()` 67.

⟨*function* `getmem_h` 72b⟩≡ (128a)
```
ushort
getmem_h(uintptr addr)
{
    byte *va;
    ulong w;

    if(addr&1) {
        w = getmem_h(addr & ~1);
        while(addr & 1) {
            w = (w>>8) | (w<<8);
            addr--;
        }
        return w;
    }
    ⟨getmem_x() if membpt 110d⟩

    va = page_of_vaddr(addr);
    va += addr&(BY2PG-1);

    return va[1]<<8 | va[0];
}
```
Uses `BY2PG` 33e, `getmem_h()` 72b, and `page_of_vaddr()` 67.

⟨*function* `getmem_w` 72c⟩≡ (128a)
```
ulong
getmem_w(uintptr addr)
{
    byte *va;
    ulong w;

    if(addr&3) {
        w = getmem_w(addr & ~3);
        while(addr & 3) {
            w = (w>>8) | (w<<24);
            addr--;
        }
        return w;
    }
    ⟨getmem_x() if membpt 110d⟩

    va = page_of_vaddr(addr);
    va += addr&(BY2PG-1);
```

```
    return va[3]<<24 | va[2]<<16 | va[1]<<8 | va[0];
}
```

Uses BY2PG 33e, getmem_w() 72c, and page_of_vaddr() 67.

⟨*function* getmem_v 73a⟩≡ (128a)

```
uvlong
getmem_v(uintptr addr)
{
    return ((uvlong)getmem_w(addr+4) << 32) | getmem_w(addr);
}
```

Uses getmem_w() 72c.

`getmem_2` and `getmem_4` are byte-by-byte variants used by the debugger and disassembler rather than by the instruction interpreter. They build the result one byte at a time via `getmem_b()`, shifting bytes into place.

⟨*function* getmem_2 73b⟩≡ (128a)

```
ulong
getmem_2(uintptr addr)
{
    ulong val;
    int i;

    val = 0;
    for(i = 0; i < 2; i++)
        val = (val>>8) | (getmem_b(addr++)<<16);
    return val;
}
```

Uses getmem_b() 72a.

⟨*function* getmem_4 73c⟩≡ (128a)

```
ulong
getmem_4(uintptr addr)
{
    ulong val;
    int i;

    val = 0;
    for(i = 0; i < 4; i++)
        val = (val>>8) | (getmem_b(addr++)<<24);
    return val;
}
```

Uses getmem_b() 72a.

6.7 putmem_xxx()

The `putmem` family is the write counterpart to `getmem`. Unlike the getmem functions, the putmem variants for half-words and words do *not* handle unaligned addresses—they report an alignment error instead. This is stricter than the read side, matching the ARM architecture where unaligned stores trap.

⟨*function* putmem_b 73d⟩≡ (128a)

```
void
putmem_b(uintptr addr, byte data)
{
    byte *va;

    va = page_of_vaddr(addr);
    va += addr&(BY2PG-1);
```

```
    va[0] = data;

    ⟨putmem_x() if membpt 110e⟩
}
```

Uses BY2PG 33e and page_of_vaddr() 67.

⟨*function* putmem_h 74a⟩≡ (128a)

```
void
putmem_h(uintptr addr, ushort data)
{
    byte *va;

    if(addr&1) {
        Bprint(bout, "Address error (Store) vaddr %.8lux\n", addr);
        longjmp(errjmp, 0);
    }

    va = page_of_vaddr(addr);
    va += addr&(BY2PG-1);

    va[1] = data>>8;
    va[0] = data;

    ⟨putmem_x() if membpt 110e⟩
}
```

Uses BY2PG 33e, bout 31d, errjmp 122b, and page_of_vaddr() 67.

⟨*function* putmem_w 74b⟩≡ (128a)

```
void
putmem_w(uintptr addr, ulong data)
{
    byte *va;

    if(addr&3) {
        Bprint(bout, "Address error (Store) vaddr %.8lux\n", addr);
        longjmp(errjmp, 0);
    }

    va = page_of_vaddr(addr);
    va += addr&(BY2PG-1);

    va[3] = data>>24;
    va[2] = data>>16;
    va[1] = data>>8;
    va[0] = data;

    ⟨putmem_x() if membpt 110e⟩
}
```

Uses BY2PG 33e, bout 31d, errjmp 122b, and page_of_vaddr() 67.

⟨*function* putmem_v 74c⟩≡ (128a)

```
void
putmem_v(uintptr addr, uvlong data)
{
    putmem_w(addr, data); /* two stages, to catch brkchk */
    putmem_w(addr+4, data>>32);
}
```

Uses putmem_w() 74b.

Chapter 7

Syscalls Emulation

Emulators can intercept the guest program at different levels: at the hardware device level (like QEMU's full-system mode), at the C library level, or at the system call level. `5i` operates at the syscall boundary—when the emulated ARM program executes a `SWI` (software interrupt), `Ssyscall()` dispatches to one of the `sys*()` functions below, which translate the Plan 9 syscall into the equivalent host OS call. This is the same approach used by QEMU's user-mode emulation (`qemu-user`). Because `5i` traps at the syscall level rather than the device level, it does not need to emulate coprocessor registers (`MCR`, `MRC`), interrupt controllers, or any other hardware.

In this chapter I will first present the syscall dispatch table and the `Ssyscall()` entry point, then the `memio()` helper used to copy data between guest and host memory, and finally the individual syscall implementations grouped by category: file I/O, memory, directory, namespace, and miscellaneous.

⟨*global* `systab` 75⟩≡ (131b)

```
void (*systab[])(void) =
{
    [NOP]  sysnop,

    [RFORK]  sysrfork,
    [EXEC]  sysexec,
    [EXITS]  sysexits,
    [AWAIT]  sysawait,

    [BRK]  sysbrk,

    [OPEN]  sysopen,
    [CLOSE]  sysclose,
    [PREAD]  syspread,
    [PWRITE] syspwrite,
    [SEEK]  sysseek,

    [CREATE] syscreate,
    [REMOVE] sysremove,
    [CHDIR]  syschdir,
    [FD2PATH] sysfd2path,
    [STAT]  sysstat,
    [FSTAT]  sysfstat,
    [WSTAT]  syswstat,
    [FWSTAT] sysfwstat,

    [BIND]  sysbind,
    [MOUNT]  sysmount,
    [UNMOUNT] sysunmount,

    [SLEEP]  syssleep,
    [ALARM]  sysalarm,
```

```
    [PIPE]  syspipe,
    [NOTIFY] sysnotify,
    [NOTED]  sysnoted,

    [SEGATTACH] syssegattach,
    [SEGDETACH] syssegdetach,
    [SEGFREE] syssegfree,
    [SEGFLUSH] syssegflush,
    [SEGBRK] syssegbrk,

    [RENDEZVOUS] sysrendezvous,

    [DUP]  sysdup,
    [FVERSION] sysfversion,
    [FAUTH]  sysfauth,

    [ERRSTR] syserrstr,
};
```

Uses sysalarm() 90c, sysawait() 88c, sysbind() 85b, sysbrk() 79b, syschdir() 85a, sysclose() 80b, syscreate() 83b, sysdup() 86a, syserrstr() 79a, sysexec() 88b, sysexits() 86c, sysfauth() 90d, sysfd2path() 84b, sysfstat() 83a, sysfversion() 90e, sysfwstat() 88e, sysmount() 90a, sysnop() 78b, sysnoted() 88f, sysnotify() 87d, sysopen() 80a, syspipe() 87b, syspread() 81a, syspwrite() 82b, sysremove() 84a, sysrendezvous() 89f, sysrfork() 88a, sysseek() 81b, syssegattach() 89a, syssegbrk() 89e, syssegdetach() 89b, syssegflush() 89d, syssegfree() 89c, syssleep() 87a, sysstat() 82c, sysunmount() 90b, and syswstat() 88d.

The `systab` array maps syscall numbers (defined in `/sys/src/libc/9syscall/sys.h`) to handler functions using C's designated initializer syntax (`[NOP] sysnop, ...`). The syscalls fall into several groups: process control (`RFORK`, `EXEC`, `EXITS`, `AWAIT`), memory (`BRK`), file I/O (`OPEN`, `CLOSE`, `PREAD`, `PWRITE`, `SEEK`), filesystem metadata (`STAT`, `CREATE`, `REMOVE`, `CHDIR`), namespace manipulation (`BIND`, `MOUNT`, `UNMOUNT`), and miscellaneous (`PIPE`, `NOTIFY`, `DUP`, `SLEEP`). A few syscalls (semaphores) are missing from this table.

The full path from a `SWI` instruction in the guest to a real host OS call crosses three dispatch tables. Here is the trip an `open("/tmp/x", 0)` takes through `5i`:

```
guest code                      emulator
----------                      --------
MOVW  $OPEN, R0
SWI   $0                  ---> ifetch() sees 0xEF000000
                               arm_class() -> op = OSWI
                               itab[OSWI].func = Ssyscall
                                 |
                                 v
                               Ssyscall(w):
                                 call = reg.r[REGARG]  (= OPEN)
                                 bounds check systab
                                 (*systab[OPEN])()
                                 |
                                 v                host OS
                               sysopen():           ------
                                 mem-copy path arg
                                 translate flags -> open(2)
                                 r[REGRET] = host fd
```

Three levels of indirection in a row: the hardware `SWI` decodes to one `itab` slot (`OSWI`) that always goes through the single `Ssyscall`[77a] thunk, which then looks up the actual handler in `systab` keyed by the syscall number

the guest left in R0. A real ARM would vector through the exception base (0x08 on classic ARM, VBAR on newer cores) into kernel mode; the syscall number convention (R0 vs #imm in the SWI word) is an ABI choice—Plan 9 picks R0, Linux ARM EABI also uses R7, and the instruction's own 24-bit immediate field is ignored by both. Because 5i is user-mode only, there is no stack switch, no SPSR save, no return-from-exception; Ssyscall() is an ordinary C function call.

Ssyscall() is the entry point for all system calls—it is registered as the handler for CSYSCALL in the instruction table. The syscall number is passed in REGARG (R0 in Plan 9's ARM calling convention). After bounds-checking, the function dispatches through systab with an indirect call.

⟨*function* Ssyscall 77a⟩≡ (131b)
```
void
Ssyscall(instruction _unused)
{
    int call;
    USED(_unused);

    call = reg.r[REGARG];

    if(call < 0 || call >= nelem(systab) || systab[call] == nil) {
        Bprint(bout, "bad system call %d (%#ux)\n", call, call);
        dumpreg();
        Bflush(bout);
        return;
    }

    if(trace)
        itrace("SWI\t%s", sysctab[call]);

    // dispatch!
    (*systab[call])();

    Bflush(bout);
}
```
Uses REGARG 28a, bout 31d, dumpreg() 103a, itrace() 120b, reg 27d, sysctab, systab 75, and trace 120a.

7.1 memio()

memio(mb, mem, size, dir) copies size bytes between mb, a regular C pointer in 5i's own address space, and mem, a virtual address in the emulated ARM program's address space (accessed byte by byte via getmem_b()/putmem_b The dir parameter selects the direction: MemRead copies from emulated memory to mb, MemReadstring does the same but stops at a null terminator, and MemWrite goes the other direction. If mb is nil, memio() allocates a buffer.

For example, when sysopen() needs the filename, the emulated program has it at some ARM virtual address like 0x2040; sysopen() calls memio(file, 0x2040, 1024, MemReadstring) which reads byte by byte via getmem_b(0x2040), getmem_b(0x2041), etc., into 5i's local char file[1024] that it can then pass to the host open().

⟨*enum* memxxx 77b⟩≡ (124)
```
// for memio()
enum
{
    MemRead,
    MemReadstring,
    MemWrite,
};
```

⟨*function* memio 78a⟩≡ (128a)

```
char *
memio(char *mb, uintptr mem, int size, int dir)
{
    int i;
    char *buf, c;

    if(mb == nil)
        mb = emalloc(size);

    buf = mb;
    switch(dir) {
    case MemRead:
        while(size--)
            *mb++ = getmem_b(mem++);
        break;
    case MemReadstring:
        for(;;) {
            if(size-- == 0) {
                Bprint(bout, "memio: user/kernel copy too long for arm\n");
                longjmp(errjmp, 0);
            }
            c = getmem_b(mem++);
            *mb++ = c;
            if(c == '\0')
                break;
        }
        break;
    case MemWrite:
        for(i = 0; i < size; i++)
            putmem_b(mem++, *mb++);
        break;
    default:
        fatal(false, "memio");
    }
    return buf;
}
```

Uses MemRead 77b, MemReadstring 77b, MemWrite 77b, bout 31d, emalloc() 123a, errjmp 122b, fatal() 122a, getmem_b() 72a, and putmem_b() 73d.

7.2 Nop and errstr syscalls

The individual sys*() functions all follow the same pattern: read arguments from the guest stack via getmem_w() at offsets from REGSP (the emulated stack pointer), call the corresponding host syscall, store any error in errbuf, and place the return value in REGRET (R0). The offset pattern (REGSP+4, REGSP+8, ...) reflects the Plan 9 C calling convention where arguments are pushed right-to-left on the stack.

⟨*function* sysnop 78b⟩≡ (131b)

```
void
sysnop(void)
{
    Bprint(bout, "nop system call %s\n", sysctab[reg.r[1]]);
    ⟨sysnop strace 103e⟩
}
```

Uses bout 31d, reg 27d, and sysctab.

⟨*global* errbuf 78c⟩≡ (131b)

```
char  errbuf[ERRMAX];
```

⟨*function* `syserrstr` 79a⟩≡ (131b)

```
void
syserrstr(void)
{
    ulong str;
    int n;

    str = getmem_w(reg.r[REGSP]+4);
    n = getmem_w(reg.r[REGSP]+8);
    if(sysdbg)
        itrace("errstr(0x%lux, 0x%lux)", str, n);

    if(n > strlen(errbuf)+1)
        n = strlen(errbuf)+1;
    memio(errbuf, str, n, MemWrite);
    strcpy(errbuf, "no error");
    reg.r[REGRET] = n;

}
```

Uses `MemWrite` 77b, `REGRET` 28a, `REGSP` 28a, `errbuf` 78c, `getmem_w()` 72c, `itrace()` 120b, `memio()` 78a, `reg` 27d, and `sysdbg` 103d.

7.3 Memory syscalls

`sysbrk()` grows or validates the Bss segment. It checks that the requested address is above the Data segment and below the Stack, then expands the Bss segment's page table if needed via `erealloc()`. This is the emulated equivalent of the kernel's `brk` syscall, which adjusts the program's heap limit.

⟨*function* `sysbrk` 79b⟩≡ (131b)

```
void
sysbrk(void)
{
    ulong addr, osize, nsize;
    Segment *s;

    addr = getmem_w(reg.r[REGSP]+4);
    if(sysdbg)
        itrace("brk(0x%lux)", addr);

    reg.r[REGRET] = -1;
    if(addr < memory.seg[Data].base+datasize) {
        strcpy(errbuf, "address below segment");
        return;
    }
    if(addr > memory.seg[Stack].base) {
        strcpy(errbuf, "segment too big");
        return;
    }
    s = &memory.seg[Bss];
    if(addr > s->end) {
        osize = ((s->end-s->base)/BY2PG)*sizeof(byte*);
        addr = ((addr)+(BY2PG-1))&~(BY2PG-1);
        s->end = addr;
        nsize = ((s->end-s->base)/BY2PG)*sizeof(byte*);
        s->table = erealloc(s->table, osize, nsize);
    }

    reg.r[REGRET] = 0;
}
```

Uses BY2PG 33e, Bss 28b, Data 28b, REGRET 28a, REGSP 28a, Stack 28b, datasize 35b, erealloc() 123b, errbuf 78c, getmem_w() 72c, itrace() 120b, memory 28d, reg 27d, and sysdbg 103d.

7.4 File and IO syscalls

The file I/O syscalls are straightforward proxies: they read arguments from the guest stack, call the host OS equivalent, and return the result. For calls involving buffers (like `read` and `write`), `memio()` transfers data between guest and host memory. Note the special case in `sysread()`: reads from file descriptor 0 (stdin) go through `Bgetc(bin)` line by line, providing an interactive prompt.

⟨*function* sysopen 80a⟩≡ (131b)

```
void
sysopen(void)
{
    char file[1024];
    int n;
    ulong mode, name;

    name = getmem_w(reg.r[REGSP]+4);
    mode = getmem_w(reg.r[REGSP]+8);
    memio(file, name, sizeof(file), MemReadstring);

    n = open(file, mode);
    if(n < 0)
        errstr(errbuf, sizeof errbuf);

    if(sysdbg)
        itrace("open(0x%lux='%s', 0x%lux) = %d", name, file, mode, n);

    reg.r[REGRET] = n;
};
```

Uses MemReadstring 77b, REGRET 28a, REGSP 28a, errbuf 78c, getmem_w() 72c, itrace() 120b, memio() 78a, reg 27d, and sysdbg 103d.

⟨*function* sysclose 80b⟩≡ (131b)

```
void
sysclose(void)
{
    int n;
    ulong fd;

    fd = getmem_w(reg.r[REGSP]+4);
    if(sysdbg)
        itrace("close(%d)", fd);

    n = close(fd);
    if(n < 0)
        errstr(errbuf, sizeof errbuf);
    reg.r[REGRET] = n;
}
```

Uses REGRET 28a, REGSP 28a, errbuf 78c, getmem_w() 72c, itrace() 120b, reg 27d, and sysdbg 103d.

⟨*function* sysread 80c⟩≡ (131b)

```
void
sysread(vlong offset)
{
    int fd;
```

```
    ulong size, a;
    char *buf, *p;
    int n, cnt, c;

    fd = getmem_w(reg.r[REGSP]+4);
    a = getmem_w(reg.r[REGSP]+8);
    size = getmem_w(reg.r[REGSP]+12);

    buf = emalloc(size);
    if(fd == 0) {
        print("\nstdin>>");
        p = buf;
        n = 0;
        cnt = size;
        while(cnt) {
            c = Bgetc(bin);
            if(c <= 0)
                break;
            *p++ = c;
            n++;
            cnt--;
            if(c == '\n')
                break;
        }
    }
    else
        n = pread(fd, buf, size, offset);

    if(n < 0)
        errstr(errbuf, sizeof errbuf);
    else
        memio(buf, a, n, MemWrite);

    if(sysdbg)
        itrace("read(%d, 0x%lux, %d, 0x%llx) = %d", fd, a, size, offset, n);

    free(buf);
    reg.r[REGRET] = n;
}
```

Uses MemWrite 77b, REGRET 28a, REGSP 28a, bin 31d, emalloc() 123a, errbuf 78c, getmem_w() 72c, itrace() 120b, memio() 78a, reg 27d, and sysdbg 103d.

⟨*function* syspread 81a⟩≡ (131b)

```
void
syspread(void)
{
    sysread(getmem_v(reg.r[REGSP]+16));
}
```

Uses REGSP 28a, getmem_v() 73a, reg 27d, and sysread() 80c.

⟨*function* sysseek 81b⟩≡ (131b)

```
void
sysseek(void)
{
    int fd;
    ulong mode;
    ulong retp;
    vlong v;

    retp = getmem_w(reg.r[REGSP]+4);
```

```
    fd = getmem_w(reg.r[REGSP]+8);
    v = getmem_v(reg.r[REGSP]+16);
    mode = getmem_w(reg.r[REGSP]+20);
    if(sysdbg)
        itrace("seek(%d, %lld, %d)", fd, v, mode);

    v = seek(fd, v, mode);
    if(v < 0)
        errstr(errbuf, sizeof errbuf);

    putmem_v(retp, v);
}
```

Uses REGSP 28a, errbuf 78c, getmem_v() 73a, getmem_w() 72c, itrace() 120b, putmem_v() 74c, reg 27d, and sysdbg 103d.

⟨*function* syswrite 82a⟩≡ (131b)

```
void
syswrite(vlong offset)
{
    int fd;
    ulong size, a;
    char *buf;
    int n;

    fd = getmem_w(reg.r[REGSP]+4);
    a = getmem_w(reg.r[REGSP]+8);
    size = getmem_w(reg.r[REGSP]+12);

    Bflush(bout);
    buf = memio(0, a, size, MemRead);
    n = pwrite(fd, buf, size, offset);
    if(n < 0)
        errstr(errbuf, sizeof errbuf);

    if(sysdbg)
        itrace("write(%d, %lux, %d, 0x%llx) = %d", fd, a, size, offset, n);

    free(buf);

    reg.r[REGRET] = n;
}
```

Uses MemRead 77b, REGRET 28a, REGSP 28a, bout 31d, errbuf 78c, getmem_w() 72c, itrace() 120b, memio() 78a, reg 27d, and sysdbg 103d.

⟨*function* syspwrite 82b⟩≡ (131b)

```
void
syspwrite(void)
{
    syswrite(getmem_v(reg.r[REGSP]+16));
}
```

Uses REGSP 28a, getmem_v() 73a, reg 27d, and syswrite() 82a.

⟨*function* sysstat 82c⟩≡ (131b)

```
void
sysstat(void)
{
    char nambuf[1024];
    byte buf[STATMAX];
    ulong edir, name;
    int n;
```

```
    name = getmem_w(reg.r[REGSP]+4);
    edir = getmem_w(reg.r[REGSP]+8);
    n = getmem_w(reg.r[REGSP]+12);
    memio(nambuf, name, sizeof(nambuf), MemReadstring);
    if(sysdbg)
        itrace("stat(0x%lux='%s', 0x%lux, 0x%lux)", name, nambuf, edir, n);
    if(n > sizeof buf)
        errstr(errbuf, sizeof errbuf);
    else{
        n = stat(nambuf, buf, n);
        if(n < 0)
            errstr(errbuf, sizeof errbuf);
        else
            memio((char*)buf, edir, n, MemWrite);
    }
    reg.r[REGRET] = n;
}
```

Uses MemReadstring 77b, MemWrite 77b, REGRET 28a, REGSP 28a, errbuf 78c, getmem_w() 72c, itrace() 120b, memio() 78a, reg 27d, and sysdbg 103d.

⟨*function* sysfstat 83a⟩≡ (131b)

```
void
sysfstat(void)
{
    byte buf[STATMAX];
    ulong edir;
    int n, fd;

    fd = getmem_w(reg.r[REGSP]+4);
    edir = getmem_w(reg.r[REGSP]+8);
    n = getmem_w(reg.r[REGSP]+12);
    if(sysdbg)
        itrace("fstat(%d, 0x%lux, 0x%lux)", fd, edir, n);

    reg.r[REGRET] = -1;
    if(n > sizeof buf){
        strcpy(errbuf, "stat buffer too big");
        return;
    }
    n = fstat(fd, buf, n);
    if(n < 0)
        errstr(errbuf, sizeof errbuf);
    else
        memio((char*)buf, edir, n, MemWrite);
    reg.r[REGRET] = n;
}
```

Uses MemWrite 77b, REGRET 28a, REGSP 28a, errbuf 78c, getmem_w() 72c, itrace() 120b, memio() 78a, reg 27d, and sysdbg 103d.

7.5 Directory syscalls

The directory and metadata syscalls (`create`, `remove`, `stat`, `fstat`, `chdir`, `fd2path`) are similarly proxied to the host. String arguments (filenames, paths) are copied from guest memory using `memio()` with `MemReadstring`. Structured results like `stat` buffers are copied back with `MemWrite`.

⟨*function* syscreate 83b⟩≡ (131b)

```
void
syscreate(void)
```

```
{
    char file[1024];
    int n;
    ulong mode, name, perm;

    name = getmem_w(reg.r[REGSP]+4);
    mode = getmem_w(reg.r[REGSP]+8);
    perm = getmem_w(reg.r[REGSP]+12);
    memio(file, name, sizeof(file), MemReadstring);
    if(sysdbg)
        itrace("create(0x%lux='%s', 0x%lux, 0x%lux)", name, file, mode, perm);

    n = create(file, mode, perm);
    if(n < 0)
        errstr(errbuf, sizeof errbuf);

    reg.r[REGRET] = n;
}
```

Uses MemReadstring 77b, REGRET 28a, REGSP 28a, errbuf 78c, getmem_w() 72c, itrace() 120b, memio() 78a, reg 27d, and sysdbg 103d.

⟨*function* sysremove 84a⟩≡ (131b)

```
void
sysremove(void)
{
    char nambuf[1024];
    ulong name;
    int n;

    name = getmem_w(reg.r[REGSP]+4);
    memio(nambuf, name, sizeof(nambuf), MemReadstring);
    if(sysdbg)
        itrace("remove(0x%lux='%s')", name, nambuf);

    n = remove(nambuf);
    if(n < 0)
        errstr(errbuf, sizeof errbuf);
    reg.r[REGRET] = n;
}
```

Uses MemReadstring 77b, REGRET 28a, REGSP 28a, errbuf 78c, getmem_w() 72c, itrace() 120b, memio() 78a, reg 27d, and sysdbg 103d.

⟨*function* sysfd2path 84b⟩≡ (131b)

```
void
sysfd2path(void)
{
    int n;
    uint fd;
    ulong str;
    char buf[1024];

    fd = getmem_w(reg.r[REGSP]+4);
    str = getmem_w(reg.r[REGSP]+8);
    n = getmem_w(reg.r[REGSP]+12);
    if(sysdbg)
        itrace("fd2path(0x%lux, 0x%lux, 0x%lux)", fd, str, n);
    reg.r[1] = -1;
    if(n > sizeof buf){
        strcpy(errbuf, "buffer too big");
        return;
```

```
    }
    n = fd2path(fd, buf, sizeof buf);
    if(n < 0)
        errstr(buf, sizeof buf);
    else
        memio(errbuf, str, n, MemWrite);
    reg.r[REGRET] = n;

}
```

Uses MemWrite 77b, REGRET 28a, REGSP 28a, errbuf 78c, getmem_w() 72c, itrace() 120b, memio() 78a, reg 27d, and sysdbg 103d.

⟨*function* syschdir 85a⟩≡ (131b)

```
void
syschdir(void)
{
    char file[1024];
    int n;
    ulong name;

    name = getmem_w(reg.r[REGSP]+4);
    memio(file, name, sizeof(file), MemReadstring);
    if(sysdbg)
        itrace("chdir(0x%lux='%s', 0x%lux)", name, file);

    n = chdir(file);
    if(n < 0)
        errstr(errbuf, sizeof errbuf);

    reg.r[REGRET] = n;
}
```

Uses MemReadstring 77b, REGRET 28a, REGSP 28a, errbuf 78c, getmem_w() 72c, itrace() 120b, memio() 78a, reg 27d, and sysdbg 103d.

7.6 Namespace syscalls

Plan 9's namespace syscalls (`bind`, `mount`, `unmount`) allow programs to reshape their file namespace—a concept unique to Plan 9 with no direct UNIX equivalent. `sysbind()` proxies the host `bind()` call.

⟨*function* sysbind 85b⟩≡ (131b)

```
void
sysbind(void)
{
    ulong pname, pold, flags;
    char name[1024], old[1024];
    int n;

    pname = getmem_w(reg.r[REGSP]+4);
    pold = getmem_w(reg.r[REGSP]+8);
    flags = getmem_w(reg.r[REGSP]+12);
    memio(name, pname, sizeof(name), MemReadstring);
    memio(old, pold, sizeof(old), MemReadstring);
    if(sysdbg)
        itrace("bind(0x%lux='%s', 0x%lux='%s', 0x%lux)", name, name, old, old, flags);

    n = bind(name, old, flags);
    if(n < 0)
        errstr(errbuf, sizeof errbuf);
```

```
    reg.r[REGRET] = n;
}
```

Uses MemReadstring 77b, REGRET 28a, REGSP 28a, errbuf 78c, getmem_w() 72c, itrace() 120b, memio() 78a, reg 27d, and sysdbg 103d.

7.7 Misc syscalls

The remaining syscalls handle file descriptor duplication (`dup`), process exit (`exits`), sleeping (`sleep`), pipes (`pipe`), and note (signal) handling (`notify`). `sysexits()` is notable: instead of terminating immediately, it sets `count = 1` to single-step back to the debugger prompt, giving the user a chance to inspect final state before exit.

⟨*function* sysdup 86a⟩≡ (131b)

```
void
sysdup(void)
{
    int oldfd, newfd;
    int n;

    oldfd = getmem_w(reg.r[REGSP]+4);
    newfd = getmem_w(reg.r[REGSP]+8);
    if(sysdbg)
        itrace("dup(%d, %d)", oldfd, newfd);

    n = dup(oldfd, newfd);
    if(n < 0)
        errstr(errbuf, sizeof errbuf);
    reg.r[REGRET] = n;
}
```

Uses REGRET 28a, REGSP 28a, errbuf 78c, getmem_w() 72c, itrace() 120b, reg 27d, and sysdbg 103d.

⟨*constant* OERRLEN 86b⟩≡ (131b)

```
#define OERRLEN 64 /* compatibility; used in _stat etc. */
```

⟨*function* sysexits 86c⟩≡ (131b)

```
void
sysexits(void)
{
    char buf[OERRLEN];
    ulong str;

    str = getmem_w(reg.r[REGSP]+4);
    if(sysdbg)
        itrace("exits(0x%lux)", str);

    // single step to give opportunity to inspect before exit
    count = 1;
    if(str != 0) {
        memio(buf, str, sizeof buf, MemRead);
        Bprint(bout, "exits(%s)\n", buf);
    }
    else
        Bprint(bout, "exits(0)\n");
}
```

Uses MemRead 77b, OERRLEN-8 86b, REGSP 28a, bout 31d, count 40a, getmem_w() 72c, itrace() 120b, memio() 78a, reg 27d, and sysdbg 103d.

⟨*function* `syssleep` 87a⟩≡ (131b)

```
void
syssleep(void)
{
    ulong len;
    int n;

    len = getmem_w(reg.r[REGSP]+4);
    if(sysdbg)
        itrace("sleep(%d)", len);

    n = sleep(len);
    if(n < 0)
        errstr(errbuf, sizeof errbuf);

    reg.r[REGRET] = n;
}
```

Uses `REGRET` 28a, `REGSP` 28a, `errbuf` 78c, `getmem_w()` 72c, `itrace()` 120b, `reg` 27d, and `sysdbg` 103d.

⟨*function* `syspipe` 87b⟩≡ (131b)

```
void
syspipe(void)
{
    int n, p[2];
    ulong fd;

    fd = getmem_w(reg.r[REGSP]+4);
    if(sysdbg)
        itrace("pipe(%lux)", fd);

    n = pipe(p);
    if(n < 0)
        errstr(errbuf, sizeof errbuf);
    else {
        putmem_w(fd, p[0]);
        putmem_w(fd+4, p[1]);
    }
    reg.r[REGRET] = n;
}
```

Uses `REGRET` 28a, `REGSP` 28a, `errbuf` 78c, `getmem_w()` 72c, `itrace()` 120b, `putmem_w()` 74b, `reg` 27d, and `sysdbg` 103d.

⟨*global* `nofunc` 87c⟩≡ (131b)

```
ulong nofunc;
```

⟨*function* `sysnotify` 87d⟩≡ (131b)

```
void
sysnotify(void)
{
    nofunc = getmem_w(reg.r[REGSP]+4);
    if(sysdbg)
        itrace("notify(0x%lux)\n", nofunc);

    reg.r[REGRET] = 0;
}
```

Uses `REGRET` 28a, `REGSP` 28a, `getmem_w()` 72c, `itrace()` 120b, `nofunc` 87c, `reg` 27d, and `sysdbg` 103d.

7.8 Unsupported syscalls

The remaining syscalls—process creation (`rfork`, `exec`, `await`), advanced namespace operations (`mount`, `unmount`), segment manipulation, `rendezvous`, and authentication—are stubs that print an error and exit. Since `5i` emulates a single process, multi-process syscalls like `rfork` and `exec` are not needed for its typical use case (running standalone programs).

⟨*function* `sysrfork` 88a⟩≡ (131b)

```
void
sysrfork(void)
{
    Bprint(bout, "No system call %s\n", sysctab[reg.r[REGARG]]);
    exits(0);
}
```

Uses `REGARG` 28a, `bout` 31d, `reg` 27d, and `sysctab`.

⟨*function* `sysexec` 88b⟩≡ (131b)

```
void
sysexec(void)
{
    Bprint(bout, "No system call %s\n", sysctab[reg.r[REGARG]]);
    exits(0);
}
```

Uses `REGARG` 28a, `bout` 31d, `reg` 27d, and `sysctab`.

⟨*function* `sysawait` 88c⟩≡ (131b)

```
void
sysawait(void)
{
    Bprint(bout, "No system call %s\n", sysctab[reg.r[REGARG]]);
    exits(0);
}
```

Uses `REGARG` 28a, `bout` 31d, `reg` 27d, and `sysctab`.

⟨*function* `syswstat` 88d⟩≡ (131b)

```
void
syswstat(void)
{
    Bprint(bout, "No system call %s\n", sysctab[reg.r[REGARG]]);
    exits(0);
}
```

Uses `REGARG` 28a, `bout` 31d, `reg` 27d, and `sysctab`.

⟨*function* `sysfwstat` 88e⟩≡ (131b)

```
void
sysfwstat(void)
{
    Bprint(bout, "No system call %s\n", sysctab[reg.r[REGARG]]);
    exits(0);
}
```

Uses `REGARG` 28a, `bout` 31d, `reg` 27d, and `sysctab`.

⟨*function* `sysnoted` 88f⟩≡ (131b)

```
void
sysnoted(void)
{
    Bprint(bout, "No system call %s\n", sysctab[reg.r[REGARG]]);
    exits(0);
}
```

Uses `REGARG` 28a, `bout` 31d, `reg` 27d, and `sysctab`.

⟨*function* `syssegattach` 89a⟩≡ (131b)

```
void
syssegattach(void)
{
    Bprint(bout, "No system call %s\n", sysctab[reg.r[REGARG]]);
    exits(0);
}
```

Uses `REGARG` 28a, `bout` 31d, `reg` 27d, and `sysctab`.

⟨*function* `syssegdetach` 89b⟩≡ (131b)

```
void
syssegdetach(void)
{
    Bprint(bout, "No system call %s\n", sysctab[reg.r[REGARG]]);
    exits(0);
}
```

Uses `REGARG` 28a, `bout` 31d, `reg` 27d, and `sysctab`.

⟨*function* `syssegfree` 89c⟩≡ (131b)

```
void
syssegfree(void)
{
    Bprint(bout, "No system call %s\n", sysctab[reg.r[REGARG]]);
    exits(0);
}
```

Uses `REGARG` 28a, `bout` 31d, `reg` 27d, and `sysctab`.

⟨*function* `syssegflush` 89d⟩≡ (131b)

```
void
syssegflush(void)
{
    Bprint(bout, "No system call %s\n", sysctab[reg.r[REGARG]]);
    exits(0);
}
```

Uses `REGARG` 28a, `bout` 31d, `reg` 27d, and `sysctab`.

⟨*function* `syssegbrk` 89e⟩≡ (131b)

```
void
syssegbrk(void)
{
    Bprint(bout, "No system call %s\n", sysctab[reg.r[REGARG]]);
    exits(0);
}
```

Uses `REGARG` 28a, `bout` 31d, `reg` 27d, and `sysctab`.

⟨*function* `sysrendezvous` 89f⟩≡ (131b)

```
void
sysrendezvous(void)
{
    Bprint(bout, "No system call %s\n", sysctab[reg.r[REGARG]]);
    exits(0);
}
```

Uses `REGARG` 28a, `bout` 31d, `reg` 27d, and `sysctab`.

⟨*function* `sysmount` 90a⟩≡ (131b)

```
void
sysmount(void)
{
    Bprint(bout, "No system call %s\n", sysctab[reg.r[REGARG]]);
    exits(0);
}
```

Uses `REGARG` 28a, `bout` 31d, `reg` 27d, and `sysctab`.

⟨*function* `sysunmount` 90b⟩≡ (131b)

```
void
sysunmount(void)
{
    Bprint(bout, "No system call %s\n", sysctab[reg.r[REGARG]]);
    exits(0);
}
```

Uses `REGARG` 28a, `bout` 31d, `reg` 27d, and `sysctab`.

⟨*function* `sysalarm` 90c⟩≡ (131b)

```
void
sysalarm(void)
{
    Bprint(bout, "No system call %s\n", sysctab[reg.r[REGARG]]);
    exits(0);
}
```

Uses `REGARG` 28a, `bout` 31d, `reg` 27d, and `sysctab`.

⟨*function* `sysfauth` 90d⟩≡ (131b)

```
void
sysfauth(void)
{
    Bprint(bout, "No system call %s\n", sysctab[reg.r[REGARG]]);
    exits(0);
}
```

Uses `REGARG` 28a, `bout` 31d, `reg` 27d, and `sysctab`.

⟨*function* `sysfversion` 90e⟩≡ (131b)

```
void
sysfversion(void)
{
    Bprint(bout, "No system call %s\n", sysctab[reg.r[REGARG]]);
    exits(0);
}
```

Uses `REGARG` 28a, `bout` 31d, `reg` 27d, and `sysctab`.

Chapter 8

Debugger

The previous chapters covered the three core functions of `5i`: instruction interpretation, memory emulation, and system call proxying. With those in place, `5i` can execute ARM programs. But executing is not enough—when something goes wrong, you need to inspect and control the execution. This chapter presents `5i`'s built-in debugger, which provides exactly that capability.

8.1 Overview

`5i` includes a built-in debugger inspired by Plan 9's `db` (and ultimately by UNIX`adb`). The debugger uses a terse command syntax where a special character prefix identifies the command category: `:` for execution control (run, step, breakpoints), `$` for inspection (registers, stack trace, profiling), `?` and `/` for memory display, `=` for expression evaluation, and `>` for register modification.

In this chapter I will present the command parser, the two main command families (`$` for inspecting and `:` for controlling), the format and display system, and finally the breakpoint mechanism including memory watchpoints.

⟨`cmd()` *locals* 91a⟩+≡ (37c) ◁37b 91b▷
```
static char *cmdlet = ":$?/=>"; //$
```

⟨`cmd()` *locals* 91b⟩+≡ (37c) ◁91a 91c▷
```
char buf[128];
char addr[128];
```

⟨`cmd()` *locals* 91c⟩+≡ (37c) ◁91b 91d▷
```
char lastcmd[128];
```

⟨`cmd()` *locals* 91d⟩+≡ (37c) ◁91c
```
char *a, *cp, *gotint;
int n, i;
```

The command parser reads a line from stdin, splits it into an optional address expression and a command character. If the user presses Enter on an empty line, the previous command is repeated—a convenience inherited from `adb` that makes stepping through code effortless. The address part may include a comma-separated repeat count (e.g., `main,5:s` to step 5 times from `main`).

⟨`cmd()` *read and parse command and address from user input* 91e⟩≡ (37c)
```
p = buf;
n = 0;

for(;;) {
    i = Bgetc(bin);
    if(i < 0)
        exits(0);
```

```
        *p++ = i;
        n++;
        if(i == '\n')
            break;
    }

    if(buf[0] == '\n')
        strcpy(buf, lastcmd);
    else {
        buf[n-1] = '\0';
        strcpy(lastcmd, buf);
    }

    p = buf;
    a = addr;
    for(;;) {
        p = nextc(p);
        if(*p == '\0' || strchr(cmdlet, *p))
            break;
        *a++ = *p++;
    }
    *a = '\0';

    cmdcount = 1;
    cp = strchr(addr, ',');
    if(cp != nil) {
        if(cp[1] == '#')
            cmdcount = strtoul(cp+2, &gotint, 16);
        else
            cmdcount = strtoul(cp+1, &gotint, 0);
        *cp = '\0';
    }
```

Uses `bin` 31d, `cmdcount` 112f, and `nextc()` 92a.

⟨*function* `nextc` 92a⟩≡ (131a)

```
    char*
    nextc(char *p)
    {
        while(*p && (*p == ' ' || *p == '\t') && *p != '\n')
            p++;

        if(*p == '\n')
            *p = '\0';

        return p;
    }
```

⟨`cmd()` *command cases* 92b⟩+≡ (37c) ◁37d 92c▷

```
    case '$': //$
        dollar(p+1);
        break;
```

Uses `dollar()` 95b.

⟨`cmd()` *command cases* 92c⟩+≡ (37c) ◁92b 93a▷

```
    case '/':
    case '?':
        dot = expr(addr);
        for(i = 0; i < cmdcount; i++)
            quesie(p+1);
        break;
```

Uses `cmdcount` 112f, `dot` 37a, and `expr()` 94b.

⟨cmd() *command cases* 93a⟩+≡ (37c) ◁92c 93b▷
```
case '=':
    eval(addr, p+1);
    break;
```
Uses eval() 101a.

⟨cmd() *command cases* 93b⟩+≡ (37c) ◁93a
```
case '>':
    setreg(addr, p+1);
    break;
```
Uses setreg() 102.

8.2 Interface

The debugger's interface borrows heavily from adb—the classic Unix debugger—which Plan 9's db also descends from. A command is a target address (an expression that can mix symbols, literals, and the dot register), an operator (? to dump, / to dump memory, = to evaluate, > to set, : for control, $ for inspection), and an optional count and format letter. So 100,20?X dumps twenty words starting at address 100 in hex, and main?i disassembles the instruction at main. The concision is deliberate: the debugger doubles as the REPL for stepping through 5i, so typing :s to step needs to stay three keystrokes long to be usable for more than five minutes.

⟨*global* fmt 93c⟩≡ (131a)
```
char fmt = 'X';
```
Uses fmt 93c.

⟨*global* width 93d⟩≡ (131a)
```
int width = 60;
```
Uses width 93d.

⟨*global* inc 93e⟩≡ (131a)
```
int inc;
```

⟨*function* reset 93f⟩≡ (131a)
```
void
reset(void)
{
    int i, l, m;
    Segment *s;
    Breakpoint *b;

    memset(&reg, 0, sizeof(Registers));

    for(i = 0; i > Nseg; i++) {
        s = &memory.seg[i];
        l = ((s->end-s->base)/BY2PG)*sizeof(byte*);
        for(m = 0; m < l; m++)
            if(s->table[m])
                free(s->table[m]);
        free(s->table);
    }
    free(iprof);
    memset(&memory, 0, sizeof(memory));

    for(b = bplist; b; b = b->next)
        b->done = b->count;
}
```
Uses BY2PG 33e, Nseg 28b, bplist 108a, iprof 112a, memory 28d, and reg 27d.

⟨*function* numsym 94a⟩≡ (131a)

```
char*
numsym(char *addr, ulong *val)
{
    char tsym[128], *t;
    static char *delim = "‘’<>/\\@*|-~+-/=?\n";
    Symbol s;
    char c;

    t = tsym;
    while(c = *addr) {
        if(strchr(delim, c))
            break;
        *t++ = c;
        addr++;
    }
    t[0] = '\0';

    if(strcmp(tsym, ".") == 0) {
        *val = dot;
        return addr;
    }

    if(lookup(0, tsym, &s))
        *val = s.value;
    else {
        if(tsym[0] == '#')
            *val = strtoul(tsym+1, 0, 16);
        else
            *val = strtoul(tsym, 0, 0);
    }
    return addr;
}
```

Uses dot 37a.

⟨*function* expr 94b⟩≡ (131a)

```
ulong
expr(char *addr)
{
    ulong t, t2;
    char op;

    if(*addr == '\0')
        return dot;

    addr = numsym(addr, &t);

    if(*addr == '\0')
        return t;

    addr = nextc(addr);
    op = *addr++;
    numsym(addr, &t2);
    switch(op) {
    default:
        Bprint(bout, "expr syntax\n");
        return 0;
    case '+':
        t += t2;
        break;
```

```
    case '-':
        t -= t2;
        break;
    case '%':
        t /= t2;
        break;
    case '&':
        t &= t2;
        break;
    case '|':
        t |= t2;
        break;
    }

    return t;
}
```

Uses `bout` 31d, `dot` 37a, `nextc()` 92a, and `numsym()` 94a.

⟨*function* `buildargv` 95a⟩≡ (131a)

```
int
buildargv(char *str, char **args, int max)
{
    int na = 0;

    while (na < max) {
        while((*str == ' ' || *str == '\t' || *str == '\n') && *str != '\0')
            str++;

        if(*str == '\0')
            return na;

        args[na++] = str;
        while(!(*str == ' ' || *str == '\t'|| *str == '\n') && *str != '\0')
            str++;

        if(*str == '\n')
            *str = '\0';

        if(*str == '\0')
            break;

        *str++ = '\0';
    }
    return na;
}
```

8.2.1 Inspecting: $

The `$` command family is for inspection: `$r` dumps registers, `$c` and `$C` print stack traces (with or without locals), `$b` lists breakpoints, `$q` quits, `$Q` prints profiling summaries, and `$t` controls various trace modes (instruction trace, syscall trace, call tree trace). The `$i` subcommands display detailed statistics: instruction mix, TLB hit rates, segment sizes, and per-function profiles.

⟨*function* `dollar` 95b⟩≡ (131a)

```
void
dollar(char *cp)
{
    cp = nextc(cp);
```

```
switch(*cp) {
case 'c':
    stktrace(*cp);
    break;

case 'C':
    stktrace(*cp);
    break;

case 'b':
    dobplist();
    break;

case 'r':
    dumpreg();
    break;

case 'R':
    dumpreg();

case 'f':
    dumpfreg();
    break;

case 'F':
    dumpdreg();
    break;

case 'q':
    exits(0);
    break;

case 'Q':
    isum();
    tlbsum();
    segsum();
    break;

case 't':
    cp++;
    switch(*cp) {
    case '\0':
        trace = true;
        break;
    case '0':
        trace = false;
        sysdbg = false;
        calltree = false;
        break;
    case 's':
        sysdbg = true;
        break;
    case 'i':
        trace = true;
        break;
    ⟨dollar() t cases 106b⟩
    default:
        Bprint(bout, "$t[0sic]\n"); //$
        break;
```

```
        }
        break;

    case 'i':
        cp++;
        switch(*cp) {
        default:
            Bprint(bout, "$i[itsa]\n"); //$
            break;
        case 'i':
            isum();
            break;
        case 't':
            tlbsum();
            break;
        case 's':
            segsum();
            break;
        case 'a':
            isum();
            tlbsum();
            segsum();
            iprofile();
            break;
        case 'p':
            iprofile();
            break;
        }
    default:
        Bprint(bout, "?\n");
        break;

    }
}
```

Uses bout 31d, calltree 106a, dobplist() 108d, dumpdreg() 103c, dumpfreg() 103b, dumpreg() 103a, iprofile() 115, isum() 112h, nextc() 92a, segsum() 114d, stktrace() 105e, sysdbg 103d, tlbsum() 114a, and trace 120a.

8.2.2 Controlling: :

The : command family controls execution: :r resets and runs the program (with optional arguments), :s single-steps (with optional count), :c continues from a breakpoint, :b sets a breakpoint, and :d deletes one.

⟨colon() *locals* 97a⟩≡ (38a) 98b▷

```
char tbuf[512];
```

⟨colon() *print current instruction* 97b⟩≡ (38a)

```
symoff(tbuf, sizeof(tbuf), dot, CTEXT);
Bprint(bout, tbuf);
if(fmt == 'z')
    printsource(dot);
```

Uses bout 31d, dot 37a, fmt 93c, and printsource() 104b.

⟨colon() *command which return cases* 97c⟩≡ (38a) 98a▷

```
case 'b':
    breakpoint(addr, cp+1);
    return;
```

Uses breakpoint() 109a.

⟨colon() *command which return cases* 98a⟩+≡ (38a) ◁97c
```
case 'd':
    delbpt(addr);
    return;
```
Uses delbpt() 109b.

⟨colon() *locals* 98b⟩+≡ (38a) ◁97a
```
int argc;
char *argv[100];
```

⟨colon() *command cases* 98c⟩+≡ (38a) ◁38b 98d▷
```
case 'r':
    reset();
    argc = buildargv(cp+1, argv, 100);
    initstk(argc, argv);
    count = 0;
    atbpt = false;
    run();
    break;
```
Uses atbpt 108c, buildargv() 95a, count 40a, initstk() 36, reset() 93f, and run() 40c.

⟨colon() *command cases* 98d⟩+≡ (38a) ◁98c
```
case 's':
    cp = nextc(cp+1);
    count = 0;
    if(*cp)
        count = strtoul(cp, 0, 0);
    if(count == 0)
        count = 1;
    atbpt = false;
    run();
    break;
```
Uses atbpt 108c, count 40a, nextc() 92a, and run() 40c.

8.3 Format

The / and ? commands display memory at dot using a format string. pfmt() handles a rich set of format characters inherited from adb: o/O for octal, d/D for decimal, x/X for hex, b for byte, c/C for character, s/S for string, i/I for disassembly (via machdata->das()), a for symbolic address, e for all globals, and z for source file:line. Lowercase letters operate on 2-byte values, uppercase on 4-byte.

⟨*function* pfmt 98e⟩≡ (131a)
```
int
pfmt(char fmt, int mem, ulong val)
{
    int c, i;
    Symbol s;
    char *p, ch, str[1024];

    c = 0;
    switch(fmt) {
    case 'o':
        c = Bprint(bout, "%-4lo ", mem? (ushort)getmem_2(dot): val);
        inc = 2;
        break;

    case 'O':
```

```
    c = Bprint(bout, "%-8lo ", mem? getmem_4(dot): val);
    inc = 4;
    break;

case 'q':
    c = Bprint(bout, "%-4lo ", mem? (short)getmem_2(dot): val);
    inc = 2;
    break;

case 'Q':
    c = Bprint(bout, "%-8lo ", mem? (long)getmem_4(dot): val);
    inc = 4;
    break;

case 'd':
    c = Bprint(bout, "%-5ld ", mem? (short)getmem_2(dot): val);
    inc = 2;
    break;


case 'D':
    c = Bprint(bout, "%-8ld ", mem? (long)getmem_4(dot): val);
    inc = 4;
    break;

case 'x':
    c = Bprint(bout, "#%-4lux ", mem? (long)getmem_2(dot): val);
    inc = 2;
    break;

case 'X':
    c = Bprint(bout, "#%-8lux ", mem? (long)getmem_4(dot): val);
    inc = 4;
    break;

case 'u':
    c = Bprint(bout, "%-5ld ", mem? (ushort)getmem_2(dot): val);
    inc = 2;
    break;

case 'U':
    c = Bprint(bout, "%-8ld ", mem? (ulong)getmem_4(dot): val);
    inc = 4;
    break;

case 'b':
    c = Bprint(bout, "%-3ld ", mem? getmem_b(dot): val);
    inc = 1;
    break;

case 'c':
    c = Bprint(bout, "%c ", (int)(mem? getmem_b(dot): val));
    inc = 1;
    break;

case 'C':
    ch = mem? getmem_b(dot): val;
    if(isprint(ch))
        c = Bprint(bout, "%c ", ch);
    else
```

```
            c = Bprint(bout, "\\x%.2x ", ch);
        inc = 1;
        break;

    case 's':
        i = 0;
        while(ch = getmem_b(dot+i))
            str[i++] = ch;
        str[i] = '\0';
        dot += i;
        c = Bprint(bout, "%s", str);
        inc = 0;
        break;

    case 'S':
        i = 0;
        while(ch = getmem_b(dot+i))
            str[i++] = ch;
        str[i] = '\0';
        dot += i;
        for(p = str; *p; p++)
            if(isprint(*p))
                c += Bprint(bout, "%c", *p);
            else
                c += Bprint(bout, "\\x%.2ux", *p);
        inc = 0;
        break;

    case 'Y':
        p = ctime(mem? getmem_b(dot): val);
        p[30] = '\0';
        c = Bprint(bout, "%s", p);
        inc = 4;
        break;

    case 'a':
        symoff(str, sizeof(str), dot, CTEXT);
        c = Bprint(bout, str);
        inc = 0;
        break;

    case 'e':
        for(i = 0; globalsym(&s, i); i++)
            Bprint(bout, "%-15s #%lux\n", s.name, getmem_4(s.value));
        inc = 0;
        break;

    case 'I':
    case 'i':
        inc = machdata->das(symmap, dot, fmt, str, sizeof(str));
        if(inc < 0) {
            Bprint(bout, "5i: %r\n");
            return 0;
        }
        c = Bprint(bout, "\t%s", str);
        break;

    case 'n':
        c = width+1;
        inc = 0;
```

```
            break;

        case '-':
            c = 0;
            inc = -1;
            break;

        case '+':
            c = 0;
            inc = 1;
            break;

        case '^':
            c = 0;
            if(inc > 0)
                inc = -inc;
            break;

        case 'z':
            if(findsym(dot, CTEXT, &s))
                Bprint(bout, "  %s() ", s.name);
            printsource(dot);
            inc = 0;
            break;

        default:
            Bprint(bout, "bad modifier\n");
            return 0;
        }
        return c;
    }
```

Uses bout 31d, dot 37a, getmem_2() 73b, getmem_4() 73c, getmem_b() 72a, inc 93e, printsource() 104b, symmap 33a, and width 93d.

⟨*function* eval 101a⟩≡ (131a)

```
    void
    eval(char *addr, char *p)
    {
        ulong val;

        val = expr(addr);
        p = nextc(p);
        if(*p == '\0') {
            p[0] = fmt;
            p[1] = '\0';
        }
        pfmt(*p, 0, val);
        Bprint(bout, "\n");
    }
```

Uses bout 31d, expr() 94b, fmt 93c, nextc() 92a, and pfmt() 98e.

⟨*function* quesie 101b⟩≡ (131a)

```
    void
    quesie(char *p)
    {
        int c, count, i;
        char tbuf[512];

        c = 0;
        symoff(tbuf, sizeof(tbuf), dot, CTEXT);
```

```
    Bprint(bout, "%s?\t", tbuf);

    while(*p) {
        p = nextc(p);
        if(*p == '"') {
            for(p++; *p && *p != '"'; p++) {
                Bputc(bout, *p);
                c++;
            }
            if(*p)
                p++;
            continue;
        }
        count = 0;
        while(*p >= '0' && *p <= '9')
            count = count*10 + (*p++ - '0');
        if(count == 0)
            count = 1;
        p = nextc(p);
        if(*p == '\0') {
            p[0] = fmt;
            p[1] = '\0';
        }
        for(i = 0; i < count; i++) {
            c += pfmt(*p, 1, 0);
            dot += inc;
            if(c > width) {
                Bprint(bout, "\n");
                symoff(tbuf, sizeof(tbuf), dot, CTEXT);
                Bprint(bout, "%s?\t", tbuf);
                c = 0;
            }
        }
        fmt = *p++;
        p = nextc(p);
    }
    Bprint(bout, "\n");
}
```

⟨*function* `setreg` 102⟩≡ (131a)

```
void
setreg(char *addr, char *cp)
{
    int rn;

    dot = expr(addr);
    cp = nextc(cp);
    if(strcmp(cp, "pc") == 0) {
        reg.r[REGPC] = dot;
        return;
    }
    if(strcmp(cp, "sp") == 0) {
        reg.r[REGSP] = dot;
        return;
    }
    if(*cp++ == 'r') {
        rn = strtoul(cp, 0, 10);
        if(rn > 0 && rn < 16) {
            reg.r[rn] = dot;
            return;
```

```
        }
    }
    Bprint(bout, "bad register\n");
}
```

Uses REGPC 28a, REGSP 28a, bout 31d, dot 37a, expr() 94b, nextc() 92a, and reg 27d.

8.4 Dumpers

The dump functions print registers and floating-point state. Note that `dumpfreg()` and `dumpdreg()` are empty stubs—the ARM emulator does not implement floating-point registers (see Chapter 10).

⟨*function* dumpreg 103a⟩≡ (127a)

```
void
dumpreg(void)
{
    int i;

    Bprint(bout, "PC  #%-8lux SP  #%-8lux \n",
                reg.r[REGPC], reg.r[REGSP]);

    for(i = 0; i < 16; i++) {
        if((i%4) == 0 && i != 0)
            Bprint(bout, "\n");
        Bprint(bout, "R%-2d #%-8lux ", i, reg.r[i]);
    }
    Bprint(bout, "\n");
}
```

Uses REGPC 28a, REGSP 28a, bout 31d, and reg 27d.

⟨*function* dumpfreg 103b⟩≡ (127a)

```
void
dumpfreg(void)
{
}
```

⟨*function* dumpdreg 103c⟩≡ (127a)

```
void
dumpdreg(void)
{
}
```

8.5 Traces

`5i` supports several trace modes, all toggled via the `$t` commands: instruction trace (`$ti`) prints every executed instruction, syscall trace (`$ts`) prints every system call with arguments and return values (like `strace` on Linux), and call tree trace (`$tc`) prints function entries and returns.

8.5.1 Syscalls trace

⟨*global* sysdbg 103d⟩≡ (126b)

```
bool sysdbg;
```

⟨sysnop *strace* 103e⟩≡ (78b)

```
if(sysdbg)
    itrace("nop()");
```

Uses itrace() 120b and sysdbg 103d.

8.5.2 Stack trace

The stack trace command (`$C`) walks the call chain backwards from the current function up to `_main`, printing each frame with its parameters and source location. Walking the stack requires three pieces of information that the linker emits in the symbol table:

- the function symbol for each `T` entry, found by `findsym()X` using the saved PC;
- the `.frame` pseudo-symbol (an `m`-type entry) emitted by the linker for each function, whose value is the size of the function's stack frame (used to find the next frame);
- the parameter offsets (the `p` entries inside the function), used by `printparams()`[105b] to read each parameter from the stack at `fp + s.value + sizeof(saved_pc)`.

Here is what the ARM call stack looks like, walked from the current function up to its caller. Note that ARM's link register (`R14`) holds the return address of the *innermost* call, so the very first frame uses `reg.r[REGLINK]` for the saved PC, while deeper frames read it from memory at `*sp`:

```
high addresses
+-------------------+ <-- _main()'s frame
|   saved PC        |
+-------------------+
|   ...locals...    |
+-------------------+ <-- caller frame: SP at outer call
|   saved PC        |     (read by getmem_4(sp))
+-------------------+
|  param1, param2   |     (printparams reads these)
+-------------------+
|   ...locals...    |
+-------------------+ <-- current frame: SP now
|  saved PC (or LR) |
+-------------------+
|   ...locals...    |
+-------------------+ <-- reg.r[REGSP]
low addresses

sp += f.value     advances to caller's frame top
                  (f.value is .frame size for current func)
```

At each step, `stktrace()`[105e] prints the current function name and parameters, then advances `pc` and `sp` to the caller's frame: `pc` is read from the saved return slot in the current frame, and `sp` is bumped by the current frame's size. The loop stops when it reaches `_main` or when it cannot find a valid frame symbol (typically meaning the chain is broken).

⟨*constant* `STRINGSZ` 104a⟩≡ (128b)
```
#define STRINGSZ 128
```

⟨*function* `printsource` 104b⟩≡ (128b)
```
/*
 * print the value of dot as file:line
 */
void
printsource(long dot)
{
    char str[STRINGSZ];
```

```
    if (fileline(str, STRINGSZ, dot))
        Bprint(bout, "%s", str);
}
```
Uses STRINGSZ-5 104a and bout 31d.

⟨*function* printlocals 105a⟩≡ (128b)
```
void
printlocals(Symbol *fn, ulong fp)
{
    int i;
    Symbol s;

    s = *fn;
    for (i = 0; localsym(&s, i); i++) {
        if (s.class != CAUTO)
            continue;
        Bprint(bout, "\t%s=#%lux\n", s.name, getmem_4(fp-s.value));
    }
}
```
Uses bout 31d and getmem_4() 73c.

⟨*function* printparams 105b⟩≡ (128b)
```
void
printparams(Symbol *fn, ulong fp)
{
    int i;
    Symbol s;
    int first;

    fp += mach->szreg;   /* skip saved pc */
    s = *fn;
    for (first = i = 0; localsym(&s, i); i++) {
        if (s.class != CPARAM)
            continue;
        if (first++)
            Bprint(bout, ", ");
        Bprint(bout, "%s=#%lux", s.name, getmem_4(fp+s.value));
    }
    Bprint(bout, ") ");
}
```
Uses bout 31d and getmem_4() 73c.

⟨*constant* STARTSYM 105c⟩≡ (128b)
```
#define STARTSYM "_main"
```

⟨*constant* FRAMENAME 105d⟩≡ (128b)
```
#define FRAMENAME ".frame"
```

⟨*function* stktrace 105e⟩≡ (128b)
```
void
stktrace(int modif)
{
    ulong pc, sp;
    Symbol s, f;
    int i;
    char buf[512];

    pc = reg.r[REGPC];
```

```
    sp = reg.r[REGSP];
    i = 0;
    while (findsym(pc, CTEXT, &s)) {
        if(strcmp(STARTSYM, s.name) == 0) {
            Bprint(bout, "%s() at #%llux\n", s.name, s.value);
            break;
        }
        if (pc == s.value) /* at first instruction */
            f.value = 0;
        else if (findlocal(&s, FRAMENAME, &f) == 0)
            break;
        if (s.type == 'L' || s.type == 'l' || pc <= s.value+4)
            pc = reg.r[REGLINK];
        else pc = getmem_4(sp);
        sp += f.value;
        Bprint(bout, "%s(", s.name);
        printparams(&s, sp);
        printsource(s.value);
        Bprint(bout, " called from ");
        symoff(buf, sizeof(buf), pc-8, CTEXT);
        Bprint(bout, buf);
        printsource(pc-8);
        Bprint(bout, "\n");
        if(modif == 'C')
            printlocals(&s, sp);
        if(++i > 40){
            Bprint(bout, "(trace truncated)\n");
            break;
        }
    }
}
```

Uses FRAMENAME-7 105d, REGLINK 28a, REGPC 28a, REGSP 28a, STARTSYM-6 105c, bout 31d, getmem_4() 73c, printlocals() 105a, printparams() 105b, printsource() 104b, and reg 27d.

8.5.3 Call tree trace

The call tree trace prints function entries (when BL is executed) and returns (when the link register R14 is copied back into the program counter). It has two hooks because ARM has no dedicated CALL / RET instructions: calls are done with BL (branch-and-link), and returns are done by moving R14 into R15 using a regular data-processing instruction (typically MOV PC, LR or ADD PC, LR, \#0). The hook in Ibl()[62g] catches calls; the hook in dpex()[46d] catches the synthetic "return" by detecting any data-processing instruction whose destination is the program counter (rd == REGPC). This is one place where ARM's unified register file leaks into the emulator: in architectures with explicit CALL / RET instructions, neither hook would be needed—the trace could be added right next to the existing handlers.

⟨*global* calltree 106a⟩≡ (126b)
```
bool calltree;
```

⟨dollar() *t cases* 106b⟩≡ (95b)
```
case 'c':
    calltree = true;
    break;
```

Uses calltree 106a.

⟨dpex() *if calltree, when add operation* 106c⟩≡ (48a)
```
if(calltree && rd == REGPC && o2 == 0) {
    Symbol s;
```

```
        findsym(o1 + o2, CTEXT, &s);
        Bprint(bout, "%8lux return to %lux %s r0=%lux\n",
                    reg.r[REGPC], o1 + o2, s.name, reg.r[REGRET]);
    }
```

Uses REGPC 28a, REGRET 28a, bout 31d, calltree 106a, and reg 27d.

⟨Ibl() *if calltree* 107a⟩≡ (62g)

```
    if(calltree) {
        findsym(v, CTEXT, &s);
        Bprint(bout, "%8lux %s(", reg.r[REGPC], s.name);
        printparams(&s, reg.r[REGSP]);
        Bprint(bout, "from ");
        printsource(reg.r[REGPC]);
        Bputc(bout, '\n');
    }
```

Uses REGPC 28a, REGSP 28a, bout 31d, calltree 106a, printparams() 105b, printsource() 104b, and reg 27d.

8.6 Breakpoints

The breakpoint system supports both code breakpoints (stop when the PC reaches an address) and memory watchpoints (stop on read, write, or access to a data address). The `Equal` type is a conditional watchpoint that triggers when a memory location equals a specific value. Breakpoints are stored in a linked list and checked on every instruction fetch (code breakpoints) or memory access (data watchpoints, via the `membpt` flag in Chapter 6).

Note how different this is from a hardware debugger. A real debugger like DEBUGGER book [Pad16c]'s `acid` cannot afford to check a list on every instruction—that would slow the target by orders of magnitude. Instead, it patches the instruction at the breakpoint address with a trap instruction (`BKPT` on ARM, `INT 3` on x86), which costs nothing until the breakpoint fires. Watchpoints in real debuggers are even harder—they require either a small number of hardware debug registers (with strict alignment and count limits) or expensive page-fault trickery. `5i` does not have this problem. Because every instruction already goes through the interpreter loop, adding a list check costs almost nothing relative to the simulation overhead, and watchpoints are just another check inside `getmem_x()X` / `putmem_x()X`. There is no limit on the number of watchpoints, no alignment restriction, and watchpoints can fire on any byte of the address space. This is one of the underrated benefits of emulation: features that are hard or impossible on real hardware become trivial in software.

⟨*enum* breakpoint_kind 107b⟩≡ (124)

```
    enum breakpoint_kind
    {
        Instruction  = 1,

        Read  = 2,
        Write  = 4,
        Access  = Read|Write,

        Equal  = 4|8,
    };
```

Uses Read 107b and Write 107b.

⟨*struct* Breakpoint 107c⟩≡ (124)

```
    struct Breakpoint
    {
        //enum<breakpoint_kind>
        int  type;  /* Instruction/Read/Access/Write/Equal */

        uintptr addr;  /* Place at address */
```

```
    int  count;  /* To execute count times or value */
    int  done;  /* How many times passed through */

    // Extra
    ⟨Breakpoint extra fields 108b⟩
};
```

⟨*global* bplist 108a⟩≡ (126b)
```
// list<Breakpoint> (next = Breakpoint.next)
Breakpoint *bplist;
```

⟨Breakpoint *extra fields* 108b⟩≡ (107c)
```
Breakpoint* next;  /* Link to next one */
```

⟨*global* atbpt 108c⟩≡ (126b)
```
bool atbpt;
```

⟨*function* dobplist 108d⟩≡ (127b)
```
void
dobplist(void)
{
    Breakpoint *b;
    char buf[512];

    for(b = bplist; b; b = b->next) {
        switch(b->type) {
        case Instruction:
            Bprint(bout, "0x%lux,%d:b %d done, at ", b->addr, b->count, b->done);
            symoff(buf, sizeof(buf), b->addr, CTEXT);
            Bprint(bout, buf);
            break;

        case Access:
            Bprint(bout, "0x%lux,%d:ba %d done, at ", b->addr, b->count, b->done);
            symoff(buf, sizeof(buf), b->addr, CDATA);
            Bprint(bout, buf);
            break;

        case Read:
            Bprint(bout, "0x%lux,%d:br %d done, at ", b->addr, b->count, b->done);
            symoff(buf, sizeof(buf), b->addr, CDATA);
            Bprint(bout, buf);
            break;

        case Write:
            Bprint(bout, "0x%lux,%d:bw %d done, at ", b->addr, b->count, b->done);
            symoff(buf, sizeof(buf), b->addr, CDATA);
            Bprint(bout, buf);
            break;

        case Equal:
            Bprint(bout, "0x%lux,%d:be at ", b->addr, b->count);
            symoff(buf, sizeof(buf), b->addr, CDATA);
            Bprint(bout, buf);
            break;
        }
        Bprint(bout, "\n");
    }
}
```

Uses Access 107b, Equal 107b, Instruction 107b, Read 107b, Write 107b, bout 31d, and bplist 108a.

⟨*function* `breakpoint` 109a⟩≡ (127b)

```
void
breakpoint(char *addr, char *cp)
{
    Breakpoint *b;
    int type;

    cp = nextc(cp);
    type = Instruction;

    switch(*cp) {
    case 'r':
        membpt++;
        type = Read;
        break;
    case 'a':
        membpt++;
        type = Access;
        break;
    case 'w':
        membpt++;
        type = Write;
        break;
    case 'e':
        membpt++;
        type = Equal;
        break;
    }
    b = emalloc(sizeof(Breakpoint));
    b->addr = expr(addr);
    b->type = type;
    b->count = cmdcount;
    b->done = cmdcount;

    b->next = bplist;
    bplist = b;
}
```

Uses `Access` 107b, `Equal` 107b, `Instruction` 107b, `Read` 107b, `Write` 107b, `bplist` 108a, `cmdcount` 112f, `emalloc()` 123a, `expr()` 94b, `membpt` 110c, and `nextc()` 92a.

⟨*function* `delbpt` 109b⟩≡ (127b)

```
void
delbpt(char *addr)
{
    Breakpoint *b, **l;
    ulong baddr;

    baddr = expr(addr);
    l = &bplist;
    for(b = *l; b; b = b->next) {
        if(b->addr == baddr) {
            if(b->type != Instruction)
                membpt++;
            *l = b->next;
            free(b);
            return;
        }
        l = &b->next;
    }
```

```
        Bprint(bout, "no breakpoint\n");
    }
```

Uses Instruction 107b, bout 31d, bplist 108a, expr() 94b, and membpt 110c.

⟨*function* brkchk 110a⟩≡ (127b)

```
    void
    brkchk(ulong addr, int type)
    {
        Breakpoint *b;

        for(b = bplist; b; b = b->next) {
            if(b->addr == addr && (b->type&type)) {
                if(b->type == Equal && getmem_4(addr) == b->count) {
                    count = 1;
                    atbpt = true;
                    return;
                }
                if(--b->done == 0) {
                    b->done = b->count;
                    count = 1;
                    atbpt = true;
                    return;
                }
            }
        }
    }
```

Uses Equal 107b, atbpt 108c, bplist 108a, count 40a, and getmem_4() 73c.

8.6.1 Code breakpoint

⟨run() *check for breakpoints* 110b⟩≡ (40c)

```
    if(bplist)
        brkchk(reg.r[REGPC], Instruction);
```

Uses Instruction 107b, REGPC 28a, bplist 108a, brkchk() 110a, and reg 27d.

8.6.2 Memory breakpoint

⟨*global* membpt 110c⟩≡ (126b)

```
    bool membpt;
```

⟨getmem_x() *if membpt* 110d⟩≡ (72)

```
    if(membpt)
        brkchk(addr, Read);
```

Uses Read 107b, brkchk() 110a, and membpt 110c.

⟨putmem_x() *if membpt* 110e⟩≡ (74 73d)

```
    if(membpt)
        brkchk(addr, Write);
```

Uses Write 107b, brkchk() 110a, and membpt 110c.

Chapter 9

Profiler

The debugger in the previous chapter lets you stop and inspect a program. The profiler, presented in this chapter, lets you understand where a program spends its time—without stopping it.

Since `5i` executes every instruction in software, it can instrument execution for free: no sampling, no overhead, just increment a counter on every fetch. The profiler collects two kinds of data: per-instruction-class counts (how many ADDs, how many branches, etc.) stored in the `Inst` table, and per-address counts stored in the `iprof` array (indexed by `PC/PROFGRAN`). The `$Q` and `$i` debugger commands display this data as instruction mix summaries, TLB statistics, memory segment usage, and per-function hotspot profiles sorted by cycle count.

Like watchpoints in the previous chapter, this is something that emulation makes essentially free, but real-hardware profilers cannot afford. The two main approaches on real hardware are both lossy or expensive:

- Sampling (PROFILER book [Pad26]'s `tprof`, Linux's `perf`, macOS's `Instruments` / `DTrace`): the kernel periodically interrupts the program and records the current PC. Cheap, but only statistical—short-running functions are likely to be missed entirely.
- Instrumentation (PROFILER book [Pad26]'s `prof`, gcc's `gprof`, Valgrind's `callgrind`): the toolchain inserts extra code at function entry and exit. Precise, but distorts timings and slows the program.

Because `5i` dispatches through a function pointer table on every instruction anyway, adding a counter increment costs no extra dispatch and no extra overhead relative to the simulation itself. The profile is exact—every executed instruction contributes to its bucket—and per-instruction granularity is free. The trade-off, of course, is that the simulation runs much slower than the original program would on real hardware, so absolute timings are meaningless; only relative counts matter.

The profiling data structures are simple. Each entry in the `Inst` dispatch table carries a `count` field (total executions), a `taken` field (for branches: how often taken), and a `useddelay` field (historical, for MIPS-style delay slots). The `iprof` array covers the entire text segment at a granularity of `PROFGRAN` (4 bytes = 1 instruction), so `iprof[i]` counts how many times the instruction at address `textbase + 4*i` was executed.

⟨`Inst` *profiling fields* 111a⟩≡ (26c)
```
// profiling info
int count;
int taken;
int useddelay;
```

⟨`run()` *profile current instruction class* 111b⟩≡ (40c)
```
// profiling
reg.ip->count++;
```
Uses `reg` 27d.

⟨*function* `Percent` 111c⟩≡ (130b)
```
#define Percent(num, max) ((max)?((num)*100)/(max):0)
```

⟨*global* `iprof` 112a⟩≡ (126b)
```
ulong *iprof;
```

⟨*constant* `PROFGRAN` 112b⟩≡ (33e)
```
PROFGRAN = 4,
```

⟨`initmemory()` *Text segment initilisation* 112c⟩+≡ (34a) ◁34e
⟨`initmemory()` *iprof allocation* 112d⟩

⟨`initmemory()` *iprof allocation* 112d⟩≡ (112c)
```
iprof = emalloc(((s->end - s->base)/PROFGRAN)*sizeof(long));
```
Uses `PROFGRAN` 112b, `emalloc()` 123a, and `iprof` 112a.

⟨*global* `tables` 112e⟩≡ (130b)
```
Inst *tables[] = { itab, 0 };
```
Uses `itab` 27a.

⟨*global* `cmdcount` 112f⟩≡ (126b)
```
int cmdcount;
```

⟨*global* `nopcount` 112g⟩≡ (126b)
```
int nopcount;
```

`isum()`[112h] prints the instruction mix: for each instruction class, it shows the execution count and percentage, then summarizes totals by category (arithmetic, memory, branch, syscall). This is the output of the `$i` debugger command.

⟨*function* `isum` 112h⟩≡ (130b)
```
void
isum(void)
{
    Inst *i;
    int total, mems, arith, branch;
    int useddelay, taken, syscall;
    int pct, j;

    total = 0;
    mems = 0;
    arith = 0;
    branch = 0;
    useddelay = 0;
    taken = 0;
    syscall = 0;

    /* Compute the total so we can have percentages */
    for(i = itab; i->func; i++)
        if(i->name && i->count)
            total += i->count;

    Bprint(bout, "\nInstruction summary.\n\n");

    for(j = 0; tables[j]; j++) {
        for(i = tables[j]; i->func; i++) {
            if(i->name) {
                /* This is gross */
                if(i->count == 0)
                    continue;
                pct = Percent(i->count, total);
                if(pct != 0)
                    Bprint(bout, "%-8ud %3d%% %s\n",
```

```
                        i->count, Percent(i->count,
                        total), i->name);
                else
                    Bprint(bout, "%-8ud      %s\n",
                        i->count, i->name);


                switch(i->type) {
                case Imem:
                    mems += i->count;
                    break;
                case Iarith:
                    arith += i->count;
                    break;
                case Ibranch:
                    branch += i->count;
                    taken += i->taken;
                    useddelay += i->useddelay;
                    break;
                case Isyscall:
                    syscall += i->count;
                    break;
                default:
                    fatal(false, "isum bad stype %d\n", i->type);
                }

            }
        }
    }

    Bprint(bout, "\n%-8ud      Memory cycles\n", mems+total);
    Bprint(bout, "%-8ud %3d%% Instruction cycles\n",
            total, Percent(total, mems+total));
    Bprint(bout, "%-8ud %3d%% Data cycles\n\n",
            mems, Percent(mems, mems+total));

    Bprint(bout, "%-8ud %3d%% Arithmetic\n",
            arith, Percent(arith, total));

    Bprint(bout, "%-8ud %3d%% System calls\n",
            syscall, Percent(syscall, total));

    Bprint(bout, "%-8ud %3d%% Branches\n",
            branch, Percent(branch, total));

    Bprint(bout, "   %-8ud %3d%% Branches taken\n",
            taken, Percent(taken, branch));

    Bprint(bout, "   %-8ud %3d%% Delay slots\n",
            useddelay, Percent(useddelay, branch));

    Bprint(bout, "   %-8ud %3d%% Unused delay slots\n",
            branch-useddelay, Percent(branch-useddelay, branch));

    Bprint(bout, "%-8ud %3d%% Program total delay slots\n",
            nopcount, Percent(nopcount, total));
}
```

Uses `Iarith` 26b, `Ibranch` 26b, `Imem` 26b, `Isyscall` 26b, `Percent-1` 111c, `bout` 31d, `fatal()` 122a, `itab` 27a, `nopcount` 112g, and `tables` 112e.

⟨*function* tlbsum 114a⟩≡ (130b)
```
void
tlbsum(void)
{
    if(tlb.on == false)
        return;

    Bprint(bout, "\n\nTlb summary\n");

    Bprint(bout, "\n%-8d User entries\n", tlb.tlbsize);
    Bprint(bout, "%-8d Accesses\n", tlb.hit+tlb.miss);
    Bprint(bout, "%-8d Tlb hits\n", tlb.hit);
    Bprint(bout, "%-8d Tlb misses\n", tlb.miss);
    Bprint(bout, "%7d%% Hit rate\n", Percent(tlb.hit, tlb.hit+tlb.miss));
}
```
Uses Percent-1 111c, bout 31d, and tlb 69d.

⟨*global* stype 114b⟩≡ (130b)
```
char *stype[] = { "Stack", "Text", "Data", "Bss" };
```

⟨Segment *profiling fields* 114c⟩≡ (29)
```
int rss;
int refs;
```

⟨*function* segsum 114d⟩≡ (130b)
```
void
segsum(void)
{
    Segment *s;
    int i;

    Bprint(bout, "\n\nMemory Summary\n\n");
    Bprint(bout, "      Base     End      Resident References\n");
    for(i = 0; i < Nseg; i++) {
        s = &memory.seg[i];
        Bprint(bout, "%-5s %.8lux %.8lux %-8d %-8d\n",
                stype[i], s->base, s->end, s->rss*BY2PG, s->refs);
    }
}
```
Uses BY2PG 33e, Nseg 28b, bout 31d, memory 28d, and stype 114b.

⟨*struct* Prof 114e⟩≡ (130b)
```
struct Prof
{
    Symbol s;
    long count;
};
```

⟨*global* aprof 114f⟩≡ (130b)
```
// can't use prof, conflict with libc.h prof()
Prof aprof[5000];
```

⟨*function* profcmp 114g⟩≡ (130b)
```
int
profcmp(void *va, void *vb)
{
    Prof *a, *b;

    a = va;
    b = vb;
    return b->count - a->count;
}
```

`iprofile()`[115] builds a per-function profile from the `iprof` array. It walks the symbol table, sums the per-address counts within each function, sorts by total count (descending), and prints the result. This is the output of the `$Q` debugger command—the closest thing `5i` has to `prof(1)`.

⟨*function* `iprofile` 115⟩≡ (130b)

```
void
iprofile(void)
{
    Prof *p, *n;
    int i, b, e;
    ulong total;

    i = 0;
    p = aprof;
    if(textsym(&p->s, i) == 0)
        return;
    i++;
    for(;;) {
        n = p+1;
        if(textsym(&n->s, i) == 0)
            break;
        b = (p->s.value-textbase)/PROFGRAN;
        e = (n->s.value-textbase)/PROFGRAN;
        while(b < e)
            p->count += iprof[b++];
        i++;
        p = n;
    }

    qsort(prof, i, sizeof(Prof), profcmp);

    total = 0;
    for(b = 0; b < i; b++)
        total += aprof[b].count;

    Bprint(bout, "  cycles     %% symbol          file\n");
    for(b = 0; b < i; b++) {
        if(aprof[b].count == 0)
            continue;

        Bprint(bout, "%8ld %3ld.%ld %-15s ",
            aprof[b].count,
            100*aprof[b].count/total,
            (1000*aprof[b].count/total)%10,
            aprof[b].s.name);

        printsource(aprof[b].s.value);
        Bputc(bout, '\n');
    }
    memset(prof, 0, sizeof(Prof)*i);
}
```

Uses `PROFGRAN` 112b, `aprof` 114f, `bout` 31d, `iprof` 112a, `printsource()` 104b, `profcmp()` 114g, and `textbase` 34d.

Chapter 10

Advanced Topics

The previous chapters covered everything `5i` implements. This chapter briefly describes what `5i` does *not* implement: system and coprocessor instructions, floating-point operations, and potential optimizations. It also covers the signal handler that lets the user interrupt a running program.

10.1 System instructions

ARM system instructions (`MCR`, `MRC`) transfer data between general-purpose registers and coprocessor registers. They are used by the kernel to configure the MMU, caches, and interrupt controller. Since `5i` is a user-mode emulator, it does not implement these instructions—they would only appear in kernel code, which `5i` cannot run.

10.2 Coprocessor instructions

ARM uses a generic "coprocessor" interface for extensions: `STC/LDC` move data between memory and a coprocessor, `CDP` performs a coprocessor data operation, and `MCR/MRC` transfer between ARM and coprocessor registers. In practice, the most important coprocessor is CP15 (the system control coprocessor for MMU/cache configuration) and CP10/CP11 (the VFP floating-point unit). `5i` does not implement any of these.

10.3 Float instructions

ARM floating-point is provided by the VFP (Vector Floating Point) coprocessor, accessed through the coprocessor instruction interface. `5i` does not implement VFP instructions. The Plan 9 C compiler `5c` handles floating-point through software emulation (calling library routines), so most user-mode programs do not need hardware float support.

10.4 Signals

When the user presses the interrupt key (usually Delete in Plan 9), the host OS delivers a note to `5i`. The `catcher()` handler sets `count` to 1, which causes the interpreter loop to stop after the current instruction and return to the debugger prompt. If the user interrupts more than five times (e.g., the emulator is stuck), `5i` exits.

⟨`cmd()` *initialisation* 116⟩≡ (37c) 122c▷

```
notify(catcher);
```

Uses `catcher()` 117.

⟨*function* `catcher` 117⟩≡ (131a)

```
void
catcher(void *a, char *msg)
{
    static int hit = 0;

    hit++;
    if(hit > 5)
        exits(0);
    USED(a);
    if(strcmp(msg, "interrupt") != 0)
        noted(NDFLT);

    count = 1;
    print("5i\n");
    noted(NCONT);
}
```

Uses `count` 40a.

10.5 Optimisations

`5i` uses pure interpretation: each instruction is decoded and dispatched through a function pointer on every execution. Modern emulators use dynamic binary translation (as in QEMU) or JIT compilation to translate frequently executed blocks into native host instructions, achieving speedups of 10–100x. Adding such an optimization to `5i` would transform its architecture significantly—the interpreter loop would become a translator that emits host machine code—but the fundamental instruction semantics described in this book would remain the same.

Chapter 11

Conclusion

You now know how the Plan 9 ARM emulator `5i` works—from the `a.out` loader that maps text, data, and BSS segments into emulated memory, through the fetch-decode-execute loop that dispatches each 32-bit instruction to its handler, to the system call proxy that translates ARM `SWI` traps into host OS calls—and more generally how many emulators and CPU simulators work.

`5i` is a user-mode emulator: it interprets ARM instructions one at a time through a fetch-decode-execute loop, emulates memory with demand paging, and translates system calls to the host OS. Despite also including a built-in debugger and profiler, the entire emulator fits in roughly 3000 lines of C. The key insight is that a CPU is just a loop: fetch the instruction at the program counter, decode the opcode and operands, execute the operation, update the program counter, and repeat. `5i` makes this loop explicit in C, which is why it serves as such a clear model of what a CPU actually does.

11.1 Patterns and techniques

These techniques apply far beyond emulators:

- *Fetch-decode-execute loop*: a single loop that fetches, decodes, dispatches, and advances. This is both how real CPUs work and how every interpreter is structured—Python's `ceval.c`, the JVM, `rc`'s bytecodes—and even game engines (read input, update state, render).
- *Dispatch table*: a function-pointer array indexed by opcode replaces a giant switch with a data structure. The same pattern drives the kernel's system call table and HTTP method dispatch in web servers: extensible without modifying the loop.
- *Demand paging*: allocating memory pages lazily, only when first accessed. The same idea appears in memory-mapped files, sparse arrays, and copy-on-write data structures: never allocate what might never be used.
- *System call proxying*: translating one system interface into another. Wine translates Windows calls to Linux, WSL 1 translated Linux syscalls to Windows, FUSE translates file operations to user-space handlers. The emulator's `SWI` trap is the purest example of this Adapter pattern.

11.2 Connections to other books

- Assembler book [Pad15a]: `5a` encodes the ARM instructions that `5i` decodes. Reading the emulator alongside the assembler gives you both sides of the instruction encoding.
- Linker book [Pad15b]: `5l` produces the `a.out` executable format that `5i` loads at startup, including the text, data, and BSS segments.

- Compiler book [Pad16a]: `5c` compiles C code into the ARM instructions that `5i` executes. The emulator is the simplest way to see what compiled code actually does.
- Kernel book [Pad14]: the kernel implements the real versions of the system calls that `5i` emulates in Chapter 7. Comparing `5i`'s syscall stubs with the kernel's full implementations shows what the emulator simplifies.
- Debugger book [Pad16c]: `5i` uses `libmach` for symbol table access and disassembly, the same library that the debugger `db` uses.

11.3 Beyond the Plan 9 emulator

`5i` is a simple interpretive emulator. Modern emulation and virtualization tools use a range of more sophisticated techniques:

- *Dynamic binary translation*: QEMU translates guest instructions into host instructions at runtime, caching the translated blocks for reuse. This is far faster than interpretation—QEMU can run a full ARM Linux system on an x86 host at reasonable speed. Apple's Rosetta 2 uses the same technique to run x86 applications on Apple Silicon with near-native performance.
- *Just-in-time compilation*: emulators like QEMU, Java's HotSpot JVM, and JavaScript engines (V8, SpiderMonkey) compile hot code paths to native machine code on the fly, applying optimizations based on runtime profiling. The gap between interpretation and JIT compilation can be 10–100x in performance.
- *Hardware virtualization*: instead of emulating instructions in software, modern CPUs provide hardware support for running guest code directly (Intel VT-x, ARM Virtualization Extensions). Hypervisors like KVM and Xen use this to run unmodified guest operating systems at near-native speed, trapping only on privileged operations.
- *Full-system emulation*: `5i` emulates only user-mode code, translating system calls to the host. QEMU in system mode emulates an entire machine including peripherals, interrupt controllers, and MMU hardware, allowing it to boot a complete guest OS.
- *Architecture evolution*: the ARM architecture has moved on to ARMv8 (Aarch64), a cleaner 64-bit design that drops some of ARMv6's complexities—notably conditional execution of every instruction. Hennessy and Patterson's latest textbook uses a subset of ARMv8 called LEGv8 for this reason. The ARMv8 reference manual is over 5000 pages, a measure of how much complexity modern architectures have accumulated.

The principles remain the same: at its core, every emulator is a loop that fetches, decodes, and executes instructions. Dynamic translation and hardware virtualization are performance optimizations on top of this fundamental loop. `5i` presents the loop in its purest form.

Appendix A

Debugging

There is a certain irony in a debugging appendix for an emulator that is itself a debugger. The facilities here are for debugging `5i` *itself*—not the guest program running inside it. When you suspect that the interpreter is mis-decoding an instruction, or that the page-table code is handing out the wrong physical page, you flip the `trace` global to `true` and every executed instruction is dumped by `itrace()`[120b] with its address, encoded word, and opcode index. The formatted output reuses the disassembler tables (`shtype`[120c] for shift types, `cond`[120d] for condition codes) so the dump looks like Plan 9 ARM assembler rather than raw hex. The reason `itrace()` flushes `bout`[31d] after every line is that a subsequent fatal decode error would otherwise lose the last few thousand instructions' worth of context—an expensive lesson learned the hard way.

⟨*global* `trace` 120a⟩≡ (126b)
```
bool trace;
```

⟨*function* `itrace` 120b⟩≡ (127a)
```
void
itrace(char *fmt, ...)
{
    char buf[128];
    va_list arg;

    va_start(arg, fmt);
    vseprint(buf, buf+sizeof(buf), fmt, arg);
    va_end(arg);

    Bprint(bout, "%8lux %.8lux %2d %s\n",
                    reg.ar, reg.instr, reg.instr_opcode, buf);
    Bflush(bout);
}
```
Uses `bout` 31d and `reg` 27d.

⟨*global* `shtype` 120c⟩≡ (128c)
```
static char* shtype[4] =
{
    "<<",
    ">>",
    "->",
    "@>",
};
```

⟨*global* `cond` 120d⟩≡ (128c)
```
static char* cond[16] =
{
    ".EQ", ".NE", ".HS", ".LO",
    ".MI", ".PL", ".VS", ".VC",
    ".HI", ".LS", ".GE", ".LT",
```

```
    ".GT", ".LE", "", ".NO",
};
```

Appendix B

Error Management

Error handling in `5i` has two regimes. Fatal errors—out of memory, malformed binary, unrecoverable decode failure—go through `fatal()`[122a], which prints a diagnostic to `STDERR` and calls `exits()`. Recoverable errors—the guest program executes an undefined instruction, or the REPL parses a bad command—go through the global `errjmp`[122b] using `setjmp`/`longjmp`. `cmd()`[37c] installs the `setjmp` checkpoint once at the start of its REPL loop; any `longjmp(errjmp, 0)` anywhere in the interpreter unwinds back to the prompt, giving the user another chance to inspect state and try again. The same pattern appears in `ed` and the Plan 9 shell (see the EDITOR book [Pad15c] and SHELL book [Pad18]): a long-lived interactive process wraps each command in a setjmp so a deep failure drops you back to the prompt rather than killing the whole session.

⟨*function* `fatal` 122a⟩≡ (127a)
```
void
fatal(bool syserr, char *fmt, ...)
{
    char buf[ERRMAX], *s;
    va_list arg;

    va_start(arg, fmt);
    vseprint(buf, buf+sizeof(buf), fmt, arg);
    va_end(arg);

    s = "5i: %s\n";
    if(syserr)
        s = "5i: %s: %r\n";
    fprint(STDERR, s, buf);
    exits(buf);
}
```

⟨*global* `errjmp` 122b⟩≡ (126b)
```
jmp_buf errjmp;
```

⟨`cmd()` *initialisation* 122c⟩+≡ (37c) ◁116
```
setjmp(errjmp);
```
Uses `errjmp` 122b.

⟨*function* `undef` 122d⟩≡ (128c)
```
void
undef(instruction inst)
{
    Bprint(bout, "undefined instruction trap pc #%lux inst %.8lux op %d\n",
        reg.r[REGPC], inst, reg.instr_opcode);
    longjmp(errjmp, 0);
}
```
Uses `REGPC` 28a, `bout` 31d, `errjmp` 122b, and `reg` 27d.

Appendix C

Utilities

`5i` has very few utilities of its own because it leans heavily on `libc` and `libbio`: the instruction dumper uses `print` and `seprint`, the REPL uses `Bioread`, and the guest-memory copy routines use plain `memmove`. The one class of helper the emulator does define is abort-on-failure allocators (`emalloc`, `erealloc`) that wrap `malloc` and route failures through `fatal()`[122a] from the previous chapter, so the rest of the code can assume allocation always succeeds and skip per-call null checks. This is the same e-prefix convention used across the Plan 9 userland (see the WINDOWS book [Pad16d] and LIBCORE book [Pad16b]).

C.1 Memory Management

⟨*function* `emalloc` 123a⟩≡ (127a)

```
void *
emalloc(ulong size)
{
    void *a;

    a = malloc(size);
    if(a == nil)
        fatal(false, "no memory");

    memset(a, 0, size); //!!
    return a;
}
```

Uses `fatal()` 122a.

⟨*function* `erealloc` 123b⟩≡ (127a)

```
void *
erealloc(void *a, ulong oldsize, ulong size)
{
    void *n;

    n = malloc(size);
    if(n == nil)
        fatal(false, "no memory");
    memset(n, 0, size);
    if(size > oldsize)
        size = oldsize;
    memmove(n, a, size);
    return n;
}
```

Uses `fatal()` 122a.

Appendix D

Extra Code

D.1 machine/5i/

D.1.1 machine/5i/arm.h

⟨machine/5i/arm.h 124⟩≡

```
/*
 * arm.h
 */

// forward decls
typedef struct Registers Registers;
typedef struct Segment  Segment;
typedef struct Memory  Memory;
typedef struct Inst  Inst;
typedef struct Icache  Icache;
typedef struct Tlb   Tlb;
typedef struct Breakpoint Breakpoint;

⟨typedef instruction 25⟩

⟨enum breakpoint_kind 107b⟩

⟨struct Breakpoint 107c⟩

⟨enum ixxx 26b⟩

// added by pad
⟨enum opcode 26a⟩

⟨constant Nmaxtlb 69c⟩

⟨enum regxxx 28a⟩

⟨struct Tlb 69b⟩

⟨struct Icache 71a⟩


⟨struct Inst 26c⟩

⟨struct Registers 27c⟩

⟨enum memxxx 77b⟩
```

⟨*enum* compare_op 64d⟩

⟨*enum* segment_kind 28b⟩

⟨*struct* Segment 29⟩

⟨*struct* Memory 28c⟩

```
// for cmd.c
void  run(void);
// for cmd.c reset
void  initstk(int, char**);
// for syscalls.c
char*  memio(char*, ulong, int, int);
// for run.c
void  Ssyscall(ulong);
// for run.c
ulong  ifetch(ulong);
// used to be in libmach/, but I copy pasted it in run.c
//int arm_class(instruction);

void  updateicache(ulong addr);

ulong  getmem_2(ulong);
ulong  getmem_4(ulong);
uchar  getmem_b(ulong);
ushort  getmem_h(ulong);
uvlong  getmem_v(ulong);
ulong  getmem_w(ulong);
void  putmem_b(ulong, uchar);
void  putmem_h(ulong, ushort);
void  putmem_v(ulong, uvlong);
void  putmem_w(ulong, ulong);

void  cmd(void);
ulong  expr(char*);
char*  nextc(char*);


void  breakpoint(char*, char*);
void  delbpt(char*);
void  brkchk(ulong, int);
void  dobplist(void);

void  dumpdreg(void);
void  dumpfreg(void);
void  dumpreg(void);
void  stktrace(int);
void  printparams(Symbol*, ulong);
void  printsource(long);

// profiling
void  iprofile(void);
void  isum(void);
void  segsum(void);
void  tlbsum(void);

void*  emalloc(ulong);
void*  erealloc(void*, ulong, ulong);
```

```
void  fatal(int, char*, ...);
void  itrace(char*, ...);

// from libc.h
//long  lnrand(long);

/* Globals */
extern Registers reg;
extern Memory  memory;
extern Icache  icache;
extern Tlb   tlb;

extern Inst  itab[];

extern instruction dot;
extern int  count;

extern Biobuf*  bout;
extern Biobuf*  bin;
extern int  text;
extern ulong textbase;
extern int  datasize;

extern Map* symmap;

extern bool trace;
extern bool sysdbg;
extern bool calltree;
extern Breakpoint* bplist;
extern int  atbpt;
extern int  membpt;

extern jmp_buf errjmp;

extern int  cmdcount;
extern int  nopcount;
extern ulong*  iprof;
```

⟨*enum* _anon_ (machine/5i/arm.h) 7 33e⟩

Uses Breakpoint 107c, Icache 71a, Inst 26c, Memory 28c, Registers 27c, Segment 29, and Tlb 69b.

D.1.2 machine/5i/icache.c

⟨machine/5i/icache.c 126a⟩≡

```
⟨basic includes 23⟩

⟨function updateicache 71d⟩
```

D.1.3 machine/5i/globals.c

⟨machine/5i/globals.c 126b⟩≡

```
⟨basic includes 23⟩

#include <tos.h>

//in run.c
//Inst itab[];
```

⟨*global* `reg` 27d⟩
⟨*global* `memory` 28d⟩
⟨*global* `text` 31b⟩
⟨*global* `trace` 120a⟩
⟨*global* `sysdbg` 103d⟩
⟨*global* `calltree` 106a⟩
⟨*global* `icache` 71b⟩
⟨*global* `tlb` 69d⟩
⟨*global* `count` 40a⟩
⟨*global* `errjmp` 122b⟩
⟨*global* `bplist` 108a⟩
⟨*global* `atbpt` 108c⟩
⟨*global* `membpt` 110c⟩
⟨*global* `cmdcount` 112f⟩
⟨*global* `nopcount` 112g⟩
⟨*global* `dot` 37a⟩
⟨*global* `bixxx` 31d⟩
⟨*global* `iprof` 112a⟩
⟨*global* `symmap` 33a⟩
⟨*global* `datasize` 35b⟩
⟨*global* `textbase` 34d⟩

D.1.4 `machine/5i/utils.c`

⟨`machine/5i/utils.c` 127a⟩≡
⟨*basic includes* 23⟩

⟨*function* `fatal` 122a⟩

⟨*function* `itrace` 120b⟩

⟨*function* `dumpreg` 103a⟩

⟨*function* `dumpfreg` 103b⟩

⟨*function* `dumpdreg` 103c⟩

⟨*function* `emalloc` 123a⟩

⟨*function* `erealloc` 123b⟩

D.1.5 `machine/5i/bpt.c`

⟨`machine/5i/bpt.c` 127b⟩≡
⟨*basic includes* 23⟩

```
#include <ctype.h>
```

⟨*function* `dobplist` 108d⟩

⟨*function* `breakpoint` 109a⟩

⟨*function* `delbpt` 109b⟩

⟨*function* `brkchk` 110a⟩

D.1.6 machine/5i/mem.c

⟨machine/5i/mem.c 128a⟩≡
⟨*basic includes* 23⟩

```
void*  page_of_vaddr(ulong);
```

⟨*function* ifetch 70⟩

⟨*function* getmem_4 73c⟩

⟨*function* getmem_2 73b⟩

⟨*function* getmem_w 72c⟩

⟨*function* getmem_h 72b⟩

⟨*function* getmem_b 72a⟩

⟨*function* getmem_v 73a⟩

⟨*function* putmem_h 74a⟩

⟨*function* putmem_w 74b⟩

⟨*function* putmem_b 73d⟩

⟨*function* putmem_v 74c⟩

⟨*function* memio 78a⟩

⟨*function* dotlb 69g⟩

⟨*function* page_of_vaddr 67⟩

D.1.7 machine/5i/symbols.c

⟨machine/5i/symbols.c 128b⟩≡
⟨*basic includes* 23⟩

⟨*constant* STRINGSZ 104a⟩

⟨*function* printsource 104b⟩

⟨*function* printlocals 105a⟩

⟨*function* printparams 105b⟩

⟨*constant* STARTSYM 105c⟩
⟨*constant* FRAMENAME 105d⟩

⟨*function* stktrace 105e⟩

D.1.8 machine/5i/run.c

⟨machine/5i/run.c 128c⟩≡
⟨*basic includes* 23⟩

⟨*macro* XCAST 48b⟩

```
// forward decl
void undef(ulong);

void Idp0(ulong);
void Idp1(ulong);
void Idp2(ulong);
void Idp3(ulong);

void Imul(ulong);
void Imula(ulong);
void Imull(ulong);

void Iswap(ulong);
void Imem1(ulong);
void Imem2(ulong);
void Ilsm(ulong inst);

void Ib(ulong);
void Ibl(ulong);

int arm_class(instruction w);

//static int dummy;
```

⟨*global* shtype 120c⟩
⟨*global* cond 120d⟩

⟨*global* itab 27a⟩

⟨*function* runcmp 64g⟩

⟨*function* runteq 65b⟩

⟨*function* runtst 65c⟩

⟨*function* run 40c⟩

⟨*function* undef 122d⟩

⟨*function* arm_class 41⟩

⟨*function* shift 50b⟩

⟨*function* dpex 46d⟩

⟨*function* Idp0 45b⟩

⟨*function* Idp1 48d⟩

⟨*function* Idp2 49b⟩

⟨*function* Idp3 52d⟩

⟨*function* Imul 44c⟩

⟨*function* Imull 53d⟩

⟨*function* `Imula` 44d⟩

⟨*function* `Iswap` 56b⟩

⟨*function* `Imem1` 57b⟩

⟨*function* `Imem2` 59⟩

⟨*function* `Ilsm` 60e⟩

⟨*function* `Ib` 62e⟩

⟨*function* `Ibl` 62g⟩

D.1.9 `machine/5i/5i.c`

⟨`machine/5i/5i.c` 130a⟩≡
⟨*basic includes* 23⟩

```
#include <tos.h>
```

⟨*global* `file` 31a⟩
⟨*global* `bxxx` 31c⟩
⟨*global* `fhdr` 32a⟩

⟨*function* `initmemory` 34a⟩

⟨*function* `inithdr` 32b⟩

⟨*function* `initstk` 36⟩

⟨*function* `main` 31e⟩

D.1.10 `machine/5i/stats.c`

⟨`machine/5i/stats.c` 130b⟩≡
⟨*basic includes* 23⟩

⟨*function* `Percent` 111c⟩

```
typedef struct Prof Prof;
```

⟨*global* `tables` 112e⟩

⟨*function* `isum` 112h⟩

⟨*function* `tlbsum` 114a⟩

⟨*global* `stype` 114b⟩

⟨*function* `segsum` 114d⟩

⟨*struct* `Prof` 114e⟩
⟨*global* `aprof` 114f⟩

⟨*function* `profcmp` 114g⟩

⟨*function* `iprofile` 115⟩

Uses Prof 114e.

D.1.11 machine/5i/cmd.c

⟨machine/5i/cmd.c 131a⟩≡
⟨*basic includes* 23⟩

```
#include <ctype.h>
```

⟨*global* fmt 93c⟩
⟨*global* width 93d⟩
⟨*global* inc 93e⟩

⟨*function* reset 93f⟩

⟨*function* nextc 92a⟩

⟨*function* numsym 94a⟩

⟨*function* expr 94b⟩

⟨*function* buildargv 95a⟩

⟨*function* colon 38a⟩

⟨*function* dollar 95b⟩

⟨*function* pfmt 98e⟩

⟨*function* eval 101a⟩

⟨*function* quesie 101b⟩

⟨*function* catcher 117⟩

⟨*function* setreg 102⟩

⟨*function* cmd 37c⟩

D.1.12 machine/5i/syscall.c

⟨machine/5i/syscall.c 131b⟩≡
⟨*basic includes* 23⟩

```
//#define ODIRLEN 116 /* compatibility; used in _stat etc. */
```
⟨*constant* OERRLEN 86b⟩

⟨*global* errbuf 78c⟩
⟨*global* nofunc 87c⟩

```
#include "../../lib_core/libc/9syscall/sys.h"
```

⟨*global* sysctab ??⟩

⟨*function* sysnop 78b⟩

⟨*function* syserrstr 79a⟩
⟨*function* sysbind 85b⟩

⟨*function* `sysfd2path` 84b⟩

⟨*function* `syschdir` 85a⟩

⟨*function* `sysclose` 80b⟩

⟨*function* `sysdup` 86a⟩

⟨*function* `sysexits` 86c⟩

⟨*function* `sysopen` 80a⟩

⟨*function* `sysread` 80c⟩

⟨*function* `syspread` 81a⟩

⟨*function* `sysseek` 81b⟩

⟨*function* `syssleep` 87a⟩

⟨*function* `sysstat` 82c⟩

⟨*function* `sysfstat` 83a⟩

⟨*function* `syswrite` 82a⟩

⟨*function* `syspwrite` 82b⟩

⟨*function* `syspipe` 87b⟩

⟨*function* `syscreate` 83b⟩

⟨*function* `sysbrk` 79b⟩

⟨*function* `sysremove` 84a⟩

⟨*function* `sysnotify` 87d⟩

⟨*function* `sysawait` 88c⟩
⟨*function* `sysrfork` 88a⟩
⟨*function* `syswstat` 88d⟩
⟨*function* `sysfwstat` 88e⟩
⟨*function* `sysnoted` 88f⟩
⟨*function* `syssegattach` 89a⟩
⟨*function* `syssegdetach` 89b⟩
⟨*function* `syssegfree` 89c⟩
⟨*function* `syssegflush` 89d⟩
⟨*function* `sysrendezvous` 89f⟩
⟨*function* `sysunmount` 90b⟩

⟨*function* `syssegbrk` 89e⟩
⟨*function* `sysmount` 90a⟩
⟨*function* `sysalarm` 90c⟩
⟨*function* `sysexec` 88b⟩

⟨*function* `sysfauth` 90d⟩

⟨*function* `sysfversion` 90e⟩

⟨*global* `systab` 75⟩

⟨*function* `Ssyscall` 77a⟩

Glossary

```
RISC = Reduced Instruction Set Computer
CISC = Complex Instruction Set Computer
ARM  = Akorn Risc Machines
ISA  = Instruction Set Architecture
```

Indexes

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.

Access: 107b, 108d, 109a
aprof: 114f, 115
arm_class(): 40c, 41
atbpt: 38a, 38b, 98c, 98d, 108c, 110a
bi: 31c, 31e
bin: 31d, 31e, 80c, 91e
bo: 31c, 31e
bout: 31d, 31e, 37c, 38a, 64e, 64g, 65b, 65c, 67, 70, 74a, 74b, 77a, 78a, 78b, 82a, 86c, 88a, 88b, 88c, 88d, 88e, 88f, 89a, 89b, 89c, 89d, 89e, 89f, 90a, 90b, 90c, 90d, 90e, 94b, 95b, 97b, 98e, 101a, 102, 103a, 104b, 105a, 105b, 105e, 106c, 107a, 108d, 109b, 112h, 114a, 114d, 115, 120b, 122d
bplist: 93f, 108a, 108d, 109a, 109b, 110a, 110b
Breakpoint: 107c, 124
breakpoint(): 97c, 109a
Breakpoint.addr: 107c
Breakpoint.count: 107c
Breakpoint.done: 107c
Breakpoint.next: 108b
Breakpoint.type: 107c
Breakpoint (typedef): 124
breakpoint_kind: 107b
brkchk(): 110a, 110b, 110d, 110e
Bss: 28b, 35d, 69a, 79b
buildargv(): 95a, 98c
BY2PG: 33e, 34a, 34c, 35a, 35d, 35e, 67, 68a, 68b, 69a, 69g, 70, 72a, 72b, 72c, 73d, 74a, 74b, 79b, 93f, 114d
BY2WD: 33e, 36
calltree: 95b, 106a, 106b, 106c, 107a
CARITH0: 44a
CARITH1: 43f, 44a
CARITH2: 43f, 44a
CARITH3: 52a, 52b
catcher(): 116, 117
CBLOC: 60b, 60c
CBRANCH: 62b, 62c
CCcmp: 47c, 48a, 54a, 54b, 63a, 64d, 64e
CCteq: 63b, 64d, 64e
CCtst: 63b, 64d, 64e
cmd(): 31e, 37c
cmdcount: 91e, 92c, 109a, 112f

CMEM0: 55b, 55d
CMEM1: 55c, 55d
CMEM2: 55d, 55f
CMEM_BASIS: 55b, 55c, 55d, 55f
CMUL: 43d, 43e, 53b
colon(): 37d, 38a
compare_op: 64d
cond-4: 40c, 120d
count: 38b, 40a, 40c, 86c, 98c, 98d, 110a, 117
Data: 28b, 35a, 68b, 79b
datasize: 35b, 35c, 79b
delbpt(): 98a, 109b
dobplist(): 95b, 108d
dollar(): 92b, 95b
dot: 37a, 37c, 38a, 92c, 94a, 94b, 97b, 98e, 102
dotlb(): 69f, 69g
dpex(): 45b, 46d, 48d, 49b, 52d
dumpdreg(): 95b, 103c
dumpfreg(): 95b, 103b
dumpreg(): 77a, 95b, 103a
emalloc(): 34c, 35a, 35d, 35e, 68a, 68b, 69a, 78a, 80c, 109a, 112d, 123a
Equal: 107b, 108d, 109a, 110a
erealloc(): 79b, 123b
errbuf: 78c, 79a, 79b, 80a, 80b, 80c, 81b, 82a, 82c, 83a, 83b, 84a, 84b, 85a, 85b, 86a, 87a, 87b
errjmp: 67, 70, 74a, 74b, 78a, 122b, 122c, 122d
eval(): 93a, 101a
expr(): 92c, 94b, 101a, 102, 109a, 109b
fatal(): 31e, 32b, 33b, 67, 68a, 68b, 78a, 112h, 122a, 123a, 123b
fhdr: 32a, 32b, 33b, 34a, 34c, 35a, 35c
file: 31a, 31a, 31e, 36
fmt: 93c, 93c, 97b, 101a
FRAMENAME-7: 105d, 105e
getmem_2(): 73b, 98e
getmem_4(): 73c, 98e, 105a, 105b, 105e, 110a
getmem_b(): 56b, 57b, 59, 72a, 73b, 73c, 78a, 98e
getmem_h(): 59, 72b, 72b
getmem_v(): 73a, 81a, 81b, 82b
getmem_w(): 56b, 57b, 60e, 72c, 72c, 73a, 79a, 79b, 80a, 80b, 80c, 81b, 82a, 82c, 83a, 83b, 84a, 84b, 85a, 85b, 86a, 86c, 87a, 87b, 87d
Iarith: 26b, 44b, 45a, 48c, 49a, 52c, 53c, 112h
Ib(): 62d, 62e
Ibl(): 62f, 62g
Ibranch: 26b, 62d, 62f, 112h
Icache: 71a, 124
icache: 71b, 71c
Icache.hash: 71a
Icache.hashtext: 71a
Icache.lines: 71a
Icache.linesize: 71a

Icache.on: 71a
Icache.stall: 71a
Icache (typedef): 124
Idp0(): 45a, 45b
Idp1(): 48c, 48d
Idp2(): 49a, 49b
Idp3(): 52c, 52d
ifetch(): 40c, 70
Ilsm(): 60d, 60e
Imem: 26b, 56a, 56c, 57a, 58, 60d, 112h
Imem1(): 56c, 57a, 57b
Imem2(): 58, 59
Imisc: 26b, 27b
Imul(): 44b, 44c
Imula(): 44b, 44d
Imull(): 53c, 53d
inc: 93e, 98e
inithdr(): 31e, 32b
initmemory(): 31e, 34a
initstk(): 31e, 36, 98c
Inst: 26c, 124
Inst.count: 111a
Inst.func: 26c
Inst.name: 26c
Inst.taken: 111a
Inst.type: 26c
Inst.useddelay: 111a
Inst (typedef): 124
Instruction: 107b, 108d, 109a, 109b, 110b
instruction (typedef): 25
iprof: 70, 93f, 112a, 112d, 115
iprofile(): 95b, 115
isum(): 95b, 112h
Iswap(): 56a, 56b
Isyscall: 26b, 66c, 112h
itab: 27a, 40c, 112e, 112h
itrace(): 40c, 77a, 79a, 79b, 80a, 80b, 80c, 81b, 82a, 82c, 83a, 83b, 84a, 84b, 85a, 85b, 86a, 86c, 87a, 87b, 87d, 103e, 120b
main-2(): 31e
membpt: 109a, 109b, 110c, 110d, 110e
memio(): 78a, 79a, 80a, 80c, 82a, 82c, 83a, 83b, 84a, 84b, 85a, 85b, 86c
Memory: 28c, 124
memory: 28d, 34c, 35a, 35d, 35e, 67, 79b, 93f, 114d
Memory.seg: 28c
Memory (typedef): 124
MemRead: 77b, 78a, 82a, 86c
MemReadstring: 77b, 78a, 80a, 82c, 83b, 84a, 85a, 85b
MemWrite: 77b, 78a, 79a, 80c, 82c, 83a, 84b
nextc(): 38a, 91e, 92a, 94b, 95b, 98d, 101a, 102, 109a

Nmaxtlb: 69b, 69c
nofunc: 87c, 87d
nopcount: 112g, 112h
Nseg: 28b, 28c, 67, 93f, 114d
numsym(): 94a, 94b
OADC: 42, 54c
OADD: 42, 48a
OAND: 42, 46e
OB: 62a
OBIC: 42, 47a
OBL: 62a
OCMN: 42, 54a
OCMP: 42, 63a
OEOR: 42, 46e
OERRLEN-8: 86b, 86c
OLDB: 55a
OLDBU: 55e
OLDH: 55e
OLDM: 60a
OLDW: 55a
OMOV: 42, 54d
OMUL: 43c
OMULA: 43c
OMULAL: 53a
OMULALU: 53a
OMULL: 53a
OMULLU: 53a
OMVN: 42, 54d
OORR: 42, 46e
opcode: 26a
opcode_category: 26b
ORSB: 42, 54b
ORSC: 42, 54c
OSBC: 42, 54c
OSTB: 55a
OSTBU: 55e
OSTH: 55e
OSTM: 60a
OSTW: 55a
OSUB: 42, 63a
OSWI: 66a, 66b
OSWPBU: 55g, 55h
OSWPW: 55g, 55h
OTEQ: 42, 63b
OTST: 42, 63b
OUNDEF: 26a, 41, 66b
page_of_vaddr(): 67, 70, 72a, 72b, 72c, 73d, 74a, 74b
Percent-1: 111c, 112h, 114a
pfmt(): 98e, 101a

printlocals(): 105a, 105e
printparams(): 105b, 105e, 107a
printsource(): 97b, 98e, 104b, 105e, 107a, 115
Prof: 114e, 130b
Prof.count: 114e
Prof.s: 114e
Prof (typedef): 130b
profcmp(): 114g, 115
PROFGRAN: 70, 112b, 112d, 115
putmem_b(): 36, 56b, 57b, 59, 73d, 78a
putmem_h(): 59, 74a
putmem_v(): 74c, 81b
putmem_w(): 36, 56b, 57b, 60e, 74b, 74c, 87b
Read: 107b, 107b, 108d, 109a, 110d
reg: 27d, 34a, 36, 37c, 38a, 40c, 44c, 44d, 45b, 46c, 46e, 47a, 47c, 48a, 48d, 49b, 50b, 51, 52d, 53d, 54a, 54b, 54d, 56b, 57b, 59, 60e, 62e, 62g, 63a, 63b, 64b, 64e, 64g, 65b, 65c, 77a, 78b, 79a, 79b, 80a, 80b, 80c, 81a, 81b, 82a, 82b, 82c, 83a, 83b, 84a, 84b, 85a, 85b, 86a, 86c, 87a, 87b, 87d, 88a, 88b, 88c, 88d, 88e, 88f, 89a, 89b, 89c, 89d, 89e, 89f, 90a, 90b, 90c, 90d, 90e, 93f, 102, 103a, 105e, 106c, 107a, 110b, 111b, 120b, 122d
REGARG: 28a, 77a, 88a, 88b, 88c, 88d, 88e, 88f, 89a, 89b, 89c, 89d, 89e, 89f, 90a, 90b, 90c, 90d, 90e
Registers: 27c, 124
Registers.ar: 40b
Registers.cbit: 47d
Registers.cc1: 64f
Registers.cc2: 64f
Registers.compare_op: 64c
Registers.cout: 47d
Registers.instr: 40b
Registers.instr_cond: 64a
Registers.instr_opcode: 40b
Registers.ip: 40b
Registers.r: 27c
Registers (typedef): 124
REGLINK: 28a, 62g, 105e
REGPC: 28a, 34a, 37c, 38a, 40c, 44c, 44d, 46a, 46b, 46c, 50a, 53d, 57b, 59, 62e, 62g, 102, 103a, 105e, 106c, 107a, 110b, 122d
REGRET: 28a, 79a, 79b, 80a, 80b, 80c, 82a, 82c, 83a, 83b, 84a, 84b, 85a, 85b, 86a, 87a, 87b, 87d, 106c
REGSP: 28a, 36, 79a, 79b, 80a, 80b, 80c, 81a, 81b, 82a, 82b, 82c, 83a, 83b, 84a, 84b, 85a, 85b, 86a, 86c, 87a, 87b, 87d, 102, 103a, 105e, 107a
reset(): 93f, 98c
run(): 38b, 40c, 98c, 98d
runcmp(): 64e, 64g
runteq(): 64e, 65b
runtst(): 64e, 65c
Sbit: 47b, 47c, 48a, 54a, 54b, 63a, 63b
Segment: 29, 124
Segment.base: 29
Segment.end: 29
Segment.fileend: 29
Segment.fileoff: 29

Segment.refs: 114c
Segment.rss: 114c
Segment.table: 29
Segment.type: 29
Segment (typedef): 124
segment_kind: 28b
segsum(): 95b, 114d
setreg(): 93b, 102
shift(): 48d, 49b, 50b, 57b
shtype-3: 120c
SIGNBIT: 65a, 65b, 65c
Ssyscall(): 66c, 77a
Stack: 28b, 35e, 69a, 79b
STACKSIZE: 33e, 35e
STACKTOP: 33e, 35e, 36
STARTSYM-6: 105c, 105e
stktrace(): 95b, 105e
STRINGSZ-5: 104a, 104b
stype: 114b, 114d
symmap: 33a, 33b, 98e
sysalarm(): 75, 90c
sysawait(): 75, 88c
sysbind(): 75, 85b
sysbrk(): 75, 79b
syschdir(): 75, 85a
sysclose(): 75, 80b
syscreate(): 75, 83b
sysctab: 77a, 78b, 88a, 88b, 88c, 88d, 88e, 88f, 89a, 89b, 89c, 89d, 89e, 89f, 90a, 90b, 90c, 90d, 90e
sysdbg: 79a, 79b, 80a, 80b, 80c, 81b, 82a, 82c, 83a, 83b, 84a, 84b, 85a, 85b, 86a, 86c, 87a, 87b, 87d, 95b, 103d, 103e
sysdup(): 75, 86a
syserrstr(): 75, 79a
sysexec(): 75, 88b
sysexits(): 75, 86c
sysfauth(): 75, 90d
sysfd2path(): 75, 84b
sysfstat(): 75, 83a
sysfversion(): 75, 90e
sysfwstat(): 75, 88e
sysmount(): 75, 90a
sysnop(): 75, 78b
sysnoted(): 75, 88f
sysnotify(): 75, 87d
sysopen(): 75, 80a
syspipe(): 75, 87b
syspread(): 75, 81a
syspwrite(): 75, 82b
sysread(): 80c, 81a
sysremove(): 75, 84a

sysrendezvous(): 75, 89f
sysrfork(): 75, 88a
sysseek(): 75, 81b
syssegattach(): 75, 89a
syssegbrk(): 75, 89e
syssegdetach(): 75, 89b
syssegflush(): 75, 89d
syssegfree(): 75, 89c
syssleep(): 75, 87a
sysstat(): 75, 82c
systab: 75, 77a
sysunmount(): 75, 90b
syswrite(): 82a, 82b
syswstat(): 75, 88d
tables: 112e, 112h
Text: 28b, 34c, 68a
text: 31b, 31e, 68a, 68b
textbase: 34d, 34e, 70, 115
Tlb: 69b, 124
tlb: 69d, 69e, 69f, 69g, 114a
Tlb.hit: 69b
Tlb.miss: 69b
Tlb.on: 69b
Tlb.tlbent: 69b
Tlb.tlbsize: 69b
Tlb (typedef): 124
tlbsum(): 95b, 114a
trace: 40c, 77a, 95b, 120a
undef(): 27b, 44c, 44d, 53d, 54c, 60e, 64g, 65b, 65c, 122d
updateicache(): 71c, 71d
UTZERO: 33e
width: 93d, 93d, 98e
Write: 107b, 107b, 108d, 109a, 110e
__anon_enum_1: 28a
__anon_enum_2: 77b
__anon_enum_3: 33e

Bibliography

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 10

[Knu99] Donald E. Knuth. *MMIXWare, a RISC Computer for the Third Millenium*. Springer, 1999. cited page(s) 9

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. cited page(s) 10

[NS05] Noam Nisan and Shimon Shocken. *The Elements of Computing Systems*. MIT Press, 2005. cited page(s) 7, 8, 9

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 10

[Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 15, 33, 119

[Pad15a] Yoann Padioleau. *Principia Softwarica: The ARM Assembler 5a*. 2015. cited page(s) 6, 10, 21, 33, 57, 60, 118

[Pad15b] Yoann Padioleau. *Principia Softwarica: The ARM Linker 5l*. 2015. cited page(s) 6, 21, 32, 33, 62, 118

[Pad15c] Yoann Padioleau. *Principia Softwarica: The Text Editor Efuns*. 2015. cited page(s) 37, 122

[Pad16a] Yoann Padioleau. *Principia Softwarica: The Plan 9 C Compiler 5c*. 2016. cited page(s) 60, 119

[Pad16b] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 123

[Pad16c] Yoann Padioleau. *Principia Softwarica: The Plan 9 Debuggers and Tracers*. 2016. cited page(s) 20, 32, 107, 119

[Pad16d] Yoann Padioleau. *Principia Softwarica: The Plan 9 Windowing System rio*. 2016. cited page(s) 123

[Pad18] Yoann Padioleau. *Principia Softwarica: The Plan 9 Shell rc*. 2018. cited page(s) 122

[Pad26] Yoann Padioleau. *Principia Softwarica: The The Plan 9 Profilers*. 2026. cited page(s) 111

[PH13] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2013. cited page(s) 8, 10

[Tan88] Andrew S Tanenbaum. *Structured Computer Organization*. Prentice Hall, 1988. cited page(s) 10