Principia Softwarica: The ARM Emulator `5i`

version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
???

March 24, 2026

Contents

Chapter 1

Introduction

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

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
```

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] 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()`.

[5] Again I recommend [NS05] for that.

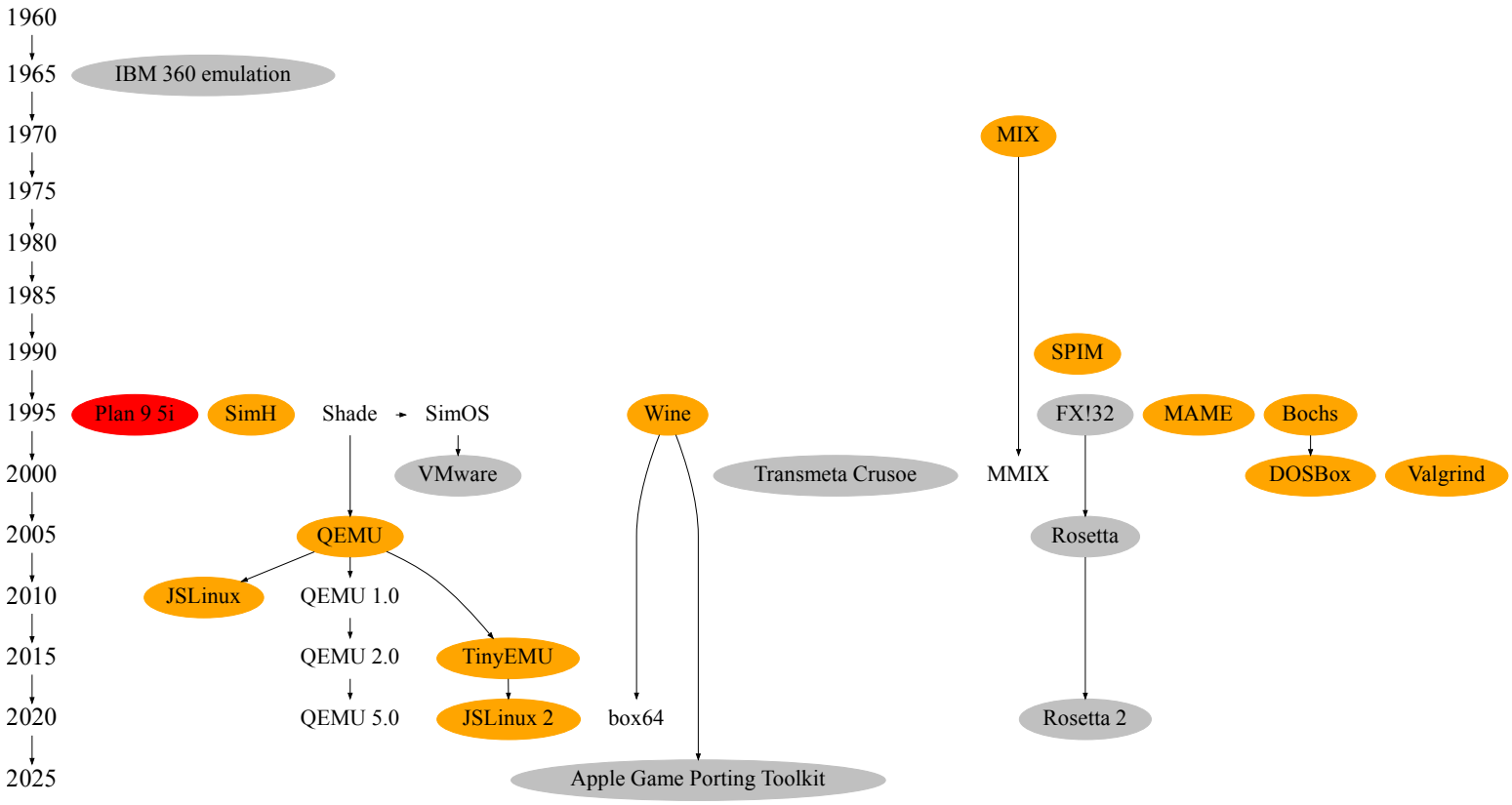


Figure 1.1: Emulators and instruction set simulators timeline

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 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

[6] http://pages.cs.wisc.edu/~larus/spim.html
[7] http://www.qemu.org
[8] http://www.mame.net/

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:

```
[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] $
```

[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

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 `c`ontinue 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.

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.

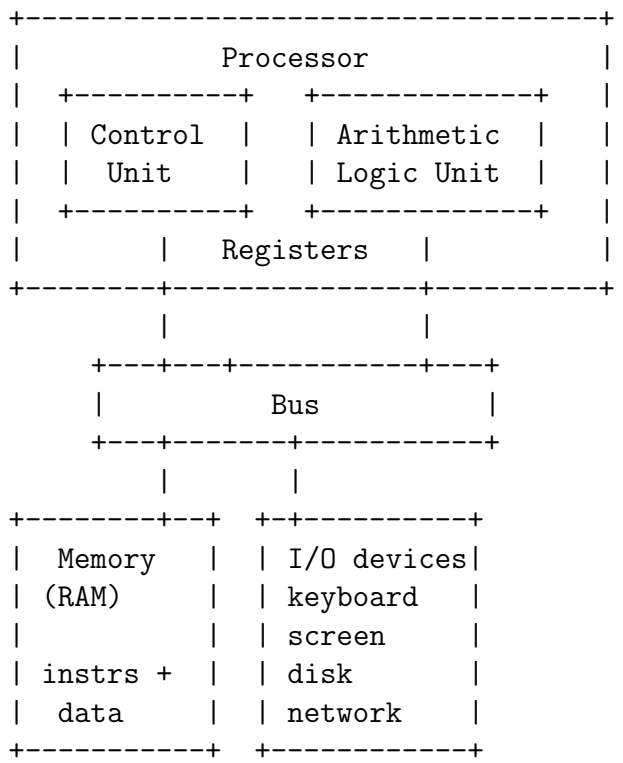
Chapter 2

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.

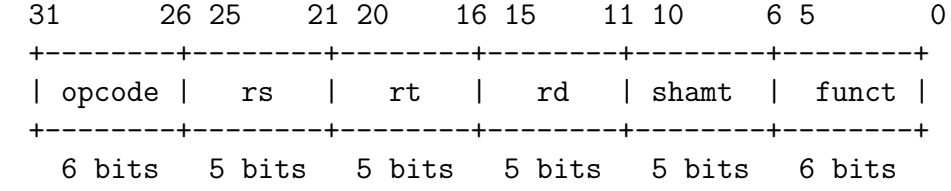
This is essentially what a universal Turing machine does—it reads a program from its tape and interprets it—except that a real processor provides a richer instruction set than a Turing machine's simple read/write/move operations.

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 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.

2.1.3 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).

2.1.4 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.

2.1.5 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.

2.1.6 Virtual memory

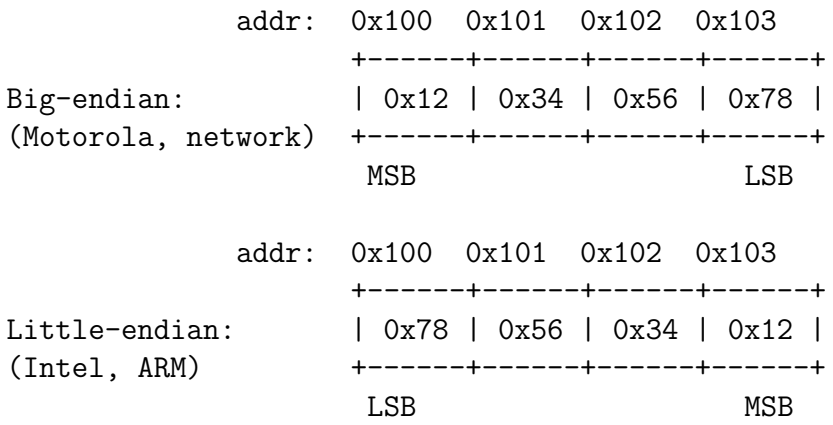
Virtual memory is a hardware mechanism that gives each process the illusion of having its own private address space. The minimal hardware support required is a TLB (Translation Lookaside Buffer) that maps virtual addresses to physical addresses, and a page fault interrupt when a mapping is missing. The 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.

2.1.7 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`:



2.1.8 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).

2.2 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. The term "emulator" (as opposed to "simulator") sometimes implies that the goal is to run real programs, not just to model timing or performance. `5i` does not simulate the pipeline, caches, or cycle counts of a real ARM processor—it only reproduces the functional behavior, which is all that matters for running programs correctly.

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 [Pad16b]). 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 15⟩≡

```
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.

[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.

Function	Ch.	File	Entities
data structures, prototypes	3	`arm.h`	`Registers`[22c] `Memory`[23b] `Icache`[63b]
entry point, loading	4	`5i.c`	`main()`[25e] `initmemory()`[28a] `initstk()`[30]
instruction interpreter	5	`run.c`	`run()`[34c] `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()`[101b]
symbol lookup	8	`symbols.c`	
debugger commands	8	`cmd.c`	`cmd()`[31c]
profiling/statistics	9	`stats.c`	`iprofile()`[106g]
globals	C	`globals.c`	
utilities	C	`utils.c`	
Total			∼3000 LOC

Table 2.1: Source files of `5i`.

```
a.out binary --> a.out header --> memory segments --> instructions --> effects
  (file)         (crackhdr)       (text, data, bss,    (fetch 4 bytes   (register
                                   stack)                + decode)        + memory
                                                                          changes)
```

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

Every source file includes the same set of headers:

⟨*basic includes* 17⟩≡ (122 121 119 118 117)

```
#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()`[25e] at the top. After loading the binary (`crackhdr()X`), initializing memory (`initmemory()`[28a]), and building the initial stack (`initstk()`[30]), `main()` enters the debugger command loop (`cmd()`[31c]). When the user types `:c`, the interpreter loop `run()`[34c] 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

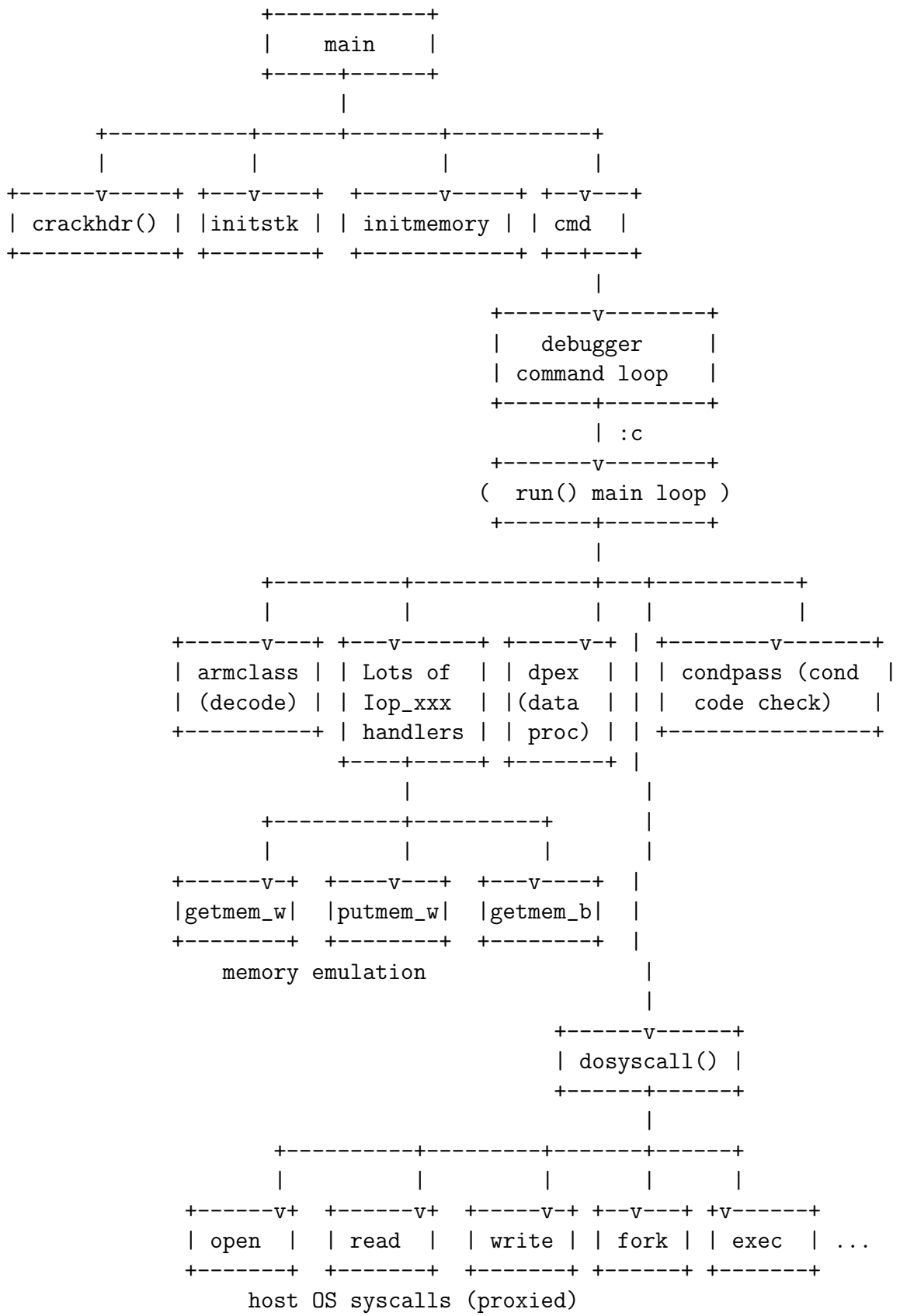


Figure 2.2: Control flow diagram of `5i`.

file directly. Load/store handlers go through the memory emulation layer (`getmem_w()`[65a], `putmem_w()`[67a], 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()`[25e], 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.

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`[21a] enumeration, and the `Inst`[21c] dispatch table that maps opcodes to handler functions. The following sections describe the machine state that `5i` simulates: the `Registers`[22c] structure (the 16 ARM registers) and the `Memory`[23b] 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* 20⟩≡ (115)
```
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`[21a] 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`[22a]:

⟨*enum* `opcode` 21a⟩≡ (115)
```
enum opcode {
    // ----------------------------------------
    // Arithmetic and logic opcodes
    // ----------------------------------------
    ⟨arith/logic opcodes 35b⟩
    // ----------------------------------------
    // Memory MOV opcodes
    // ----------------------------------------
    ⟨memory opcodes 47c⟩
    // ----------------------------------------
    // Control flow opcodes
    // ----------------------------------------
    ⟨branching opcodes 54a⟩
    // ----------------------------------------
    // Syscall opcodes
    // ----------------------------------------
    ⟨syscall opcodes 58b⟩

    // 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` 21b⟩≡ (115)
```
enum opcode_category
{
    Iarith,
    Imem,
    Ibranch,
    Isyscall,
    Imisc,
};
```

3.2 `Inst` and `itab`

Each opcode is associated with an `Inst`[21c] structure that bundles the handler function, a human-readable name (for the debugger and profiler), and the broad category:

⟨*struct* `Inst` 21c⟩≡ (115)
```
struct Inst
{
    void  (*func)(instruction);
    char* name;
    // enum<opcode_category>
    int type;

    ⟨Inst profiling fields 103a⟩
};
```

The global array `itab`[22a] maps each `opcode`[21a] 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` 22a⟩≡ (119c)
```
//map<enum<opcode>, Inst>
Inst itab[] =
{
  ⟨itab elements 22b⟩
  { 0 }
};
```

The only entry shown here is `OUNDEF`, which maps to the `undef()`[113d] error handler. The remaining 88 entries (arithmetic, memory, branches, etc.) will be filled in gradually in Chapter 5.

⟨`itab` *elements* 22b⟩≡ (22a) 37a▷
```
[OUNDEF] =  { &undef,  "UNDEF", Imisc},
```
Uses `Imisc` 21b and `undef()` 113d.

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` 22c⟩≡ (115)
```
struct Registers
{
    long r[16];
    ⟨Registers other fields 34b⟩
};
```

⟨*global* `reg` 22d⟩≡ (117b)
```
Registers reg;
```

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

⟨*enum* `regxxx` 22e⟩≡ (115)
```
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` 23a⟩≡ (115)
```
enum segment_kind
{
    Text,
    Data,
    Bss,
    Stack,

    Nseg,
};
```

⟨*struct* `Memory` 23b⟩≡ (115)
```
struct Memory
{
    //map<enum<segment_kind>, Segment>
    Segment seg[Nseg];
};
```
Uses `Nseg` 23a.

⟨*global* `memory` 23c⟩≡ (117b)
```
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` 23d⟩≡ (115)
```
struct Segment
{
    // enum<segment_kind>
    short type;
```

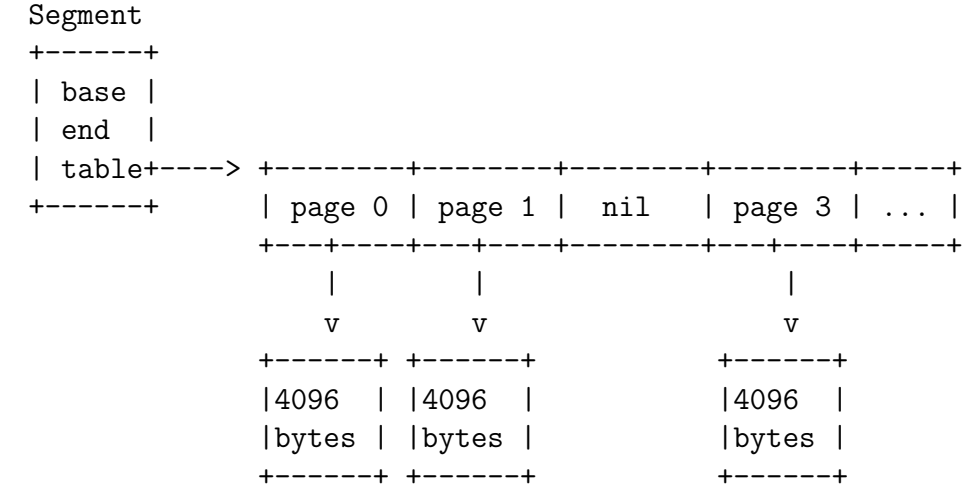
```
    uintptr base;
    uintptr end;

    //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 106b⟩
};
```

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`).



When `getmem_w()`[65a] or `putmem_b()`[66b] 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()`[60] 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()`[25e], the entry point of the emulator.

4.1 `main()` skeleton and globals

The `main()`[25e] function of `5i` performs four tasks: it opens the binary file and parses its `a.out` header (`inithdr()`[26b]), it loads the text and data segments into simulated memory (`initmemory()`[28a]), it builds the initial stack with `argc`/`argv` (`initstk()`[30]), and then it enters the debugger command loop (`cmd()`[31c]). 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` 25a⟩≡ (121a)
```
char* file = "5.out";
```
Uses `file` 25a.

The file descriptor `text` holds the opened binary. It must be a global because `page_of_vaddr()`[60] needs it later for lazy loading of text and data pages:

⟨*global* `text` 25b⟩≡ (117b)
```
fdt text;
```

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

⟨*global* `bxxx` 25c⟩≡ (121a)
```
Biobuf bi, bo;
```

⟨*global* `bixxx` 25d⟩≡ (117b)
```
Biobuf *bin, *bout;
```

The `main()` function is straightforward:

⟨*function* `main` 25e⟩≡ (121a)
```
//@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 62e⟩

    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 25c, bin 25d, bo 25c, bout 25d, cmd() 31c, fatal() 113a, file 25a, inithdr() 26b, initmemory() 28a, initstk() 30, and text 25b.

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()`[26b] function uses `libmach`'s `crackhdr()X` to parse the `a.out` header (see the DEBUGGER book [Pad16b] 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`[26a] structure, which `initmemory()`[28a] will read to know the size and offset of each segment:

⟨*global* fhdr 26a⟩≡ (121a)

```
Fhdr fhdr;
```

⟨*function* inithdr 26b⟩≡ (121a)

```
void
inithdr(fdt fd)
{
    ⟨inithdr() locals 27c⟩

    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 27b⟩
    ⟨inithdr() mach initialisation 27d⟩

}
```

Uses fatal() 113a and fhdr 26a.

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` 27a⟩≡ (117b)
```
Map *symmap;
```

⟨`inithdr()` *symmap initialisation* 27b⟩≡ (26b)
```
if (syminit(fd, &fhdr) < 0)
    fatal(false, "%r\n");

symmap = loadmap(symmap, fd, &fhdr);
```
Uses `fatal()` 113a, `fhdr` 26a, and `symmap` 27a.

⟨`inithdr()` *locals* 27c⟩≡ (26b)
```
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* 27d⟩≡ (26b)
```
//???
if (mach->sbreg && lookup(0, mach->sbreg, &s))
    mach->sb = s.value;
machdata = &armmach;
```

4.3 `initmemory()`

Now that the header is parsed, `initmemory()`[28a] 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`[26a] and creates the four segments described in Chapter 3. First, a few constants:

⟨*enum* `_anon_ (machine/5i/arm.h)` ◁ 27e⟩≡ (115)
```
enum
{
    /* Plan9 Kernel constants */
    BY2PG  = 4096,
    BY2WD  = 4,

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

    ⟨constant PROFGRAN 103e⟩
    ⟨constant Sbit 40a⟩
    ⟨constant SIGNBIT 57d⟩
};
```

`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`[23d] structures and sets the program counter to the entry point from the header:

⟨*function* `initmemory` 28a⟩≡ (121a)
```
void
initmemory(void)
{
    uintptr t, d, b, bssend;
    ⟨initmemory() locals 28b⟩

    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 28c⟩
    ⟨initmemory() Data segment initilisation 29a⟩
    ⟨initmemory() Bss segment initilisation 29d⟩
    ⟨initmemory() Stack segment initilisation 29e⟩

    reg.r[REGPC] = fhdr.entry;
}
```
Uses `BY2PG` 27e, `REGPC` 22e, `fhdr` 26a, and `reg` 22d.

⟨`initmemory()` *locals* 28b⟩≡ (28a)
```
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* 28c⟩≡ (28a) 28e▷
```
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` 27e, `Text` 23a, `emalloc()` 114a, `fhdr` 26a, and `memory` 23c.

⟨*global* `textbase` 28d⟩≡ (117b)
```
uintptr textbase;
```

⟨`initmemory()` *Text segment initilisation* 28e⟩+≡ (28a) ◁28c 103f▷
```
textbase = s->base;
```
Uses `textbase` 28d.

The data segment follows immediately after text:

⟨`initmemory()` *Data segment initilisation* 29a⟩≡ (28a) 29c▷
```
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` 27e, `Data` 23a, `emalloc()` 114a, `fhdr` 26a, and `memory` 23c.

⟨*global* `datasize` 29b⟩≡ (117b)
```
int datasize;
```

⟨`initmemory()` *Data segment initilisation* 29c⟩+≡ (28a) ◁29a
```
datasize = fhdr.datsz;
```
Uses `datasize` 29b and `fhdr` 26a.

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

⟨`initmemory()` *Bss segment initilisation* 29d⟩≡ (28a)
```
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` 27e, `Bss` 23a, `emalloc()` 114a, and `memory` 23c.

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

⟨`initmemory()` *Stack segment initilisation* 29e⟩≡ (28a)
```
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` 27e, `STACKSIZE` 27e, `STACKTOP` 27e, `Stack` 23a, `emalloc()` 114a, and `memory` 23c.

4.4 initstk()

With the four memory segments in place, the last step before entering the debugger is to populate the stack. The `initstk()`[30] 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 30⟩≡ (121a)

```
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 27e, REGSP 22e, STACKTOP 27e, file 25a, putmem_b() 66b, putmem_w() 67a, and reg 22d.

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()[31c], 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 31a⟩≡ (117b)

```
uintptr dot;
```

⟨cmd() *locals* 31b⟩≡ (31c) 84a▷

```
char *p;
```

⟨*function* cmd 31c⟩≡ (122a)

```
void
cmd(void)
{
    ⟨cmd() locals 31b⟩

    dot = reg.r[REGPC];

    ⟨cmd() initialisation 108⟩

    for(;;) {
        Bflush(bout);
        ⟨cmd() read and parse command and address from user input 84e⟩

        switch(*p) {
        ⟨cmd() command cases 31d⟩
        default:
            Bprint(bout, "?\n");
            break;
        }
    }
}
```

Uses REGPC 22e, bout 25d, dot 31a, and reg 22d.

⟨cmd() *command cases* 31d⟩≡ (31c) 85b▷

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

Uses colon() 32a.

I show `colon()`[32a] 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` 32a⟩≡ (122a)

```
void
colon(char *addr, char *cp)
{
    ⟨colon() locals 90a⟩

    cp = nextc(cp);

    switch(*cp) {
    ⟨colon() command which return cases 90c⟩
    /* These fall through to print the stopped address */
    ⟨colon() command cases 32b⟩
    default:
        Bprint(bout, "?\n");
        return;
    }

    dot = reg.r[REGPC];

    Bprint(bout, "%s at #%lux ", atbpt? "breakpoint": "stopped", dot);
    ⟨colon() print current instruction 90b⟩
    Bprint(bout, "\n");
}
```

Uses `REGPC` 22e, `atbpt` 100a, `bout` 25d, `dot` 31a, `nextc()` 85a, and `reg` 22d.

And here is the `:c` case, where we finally see the call to `run()`[34c]—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* 32b⟩≡ (32a) 91c▷

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

Uses `atbpt` 100a, `count` 34a, and `run()` 34c.

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()`[34c]) and the decoder (`arm_class()`[35a]), 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()`[35a] 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`[22a], `dpex()`[39d] dispatches on the arithmetic/logic opcode (bits 21–24: AND, ADD, ORR, ...), `shift()`[43b] 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()`[34c] 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` 34a⟩≡ (117b)
```
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()`[35a]), and `ip` points to the corresponding `Inst`[21c] 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* 34b⟩≡ (22c) 40c▷
```
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` 34c⟩≡ (119c)
```
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 56c⟩
        ⟨run() switch reg.compare_op to set execute 57a⟩

        if(execute) {
            ⟨run() profile current instruction class 103b⟩
            // !!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 102a⟩
    } while(--count);
}
```
Uses `REGPC` 22e, `arm_class()` 35a, `cond-4` 112d, `count` 34a, `ifetch()` 63a, `itab` 22a, `itrace()` 112b, `reg` 22d, and `trace` 112a.

Each iteration: fetch the instruction word from memory (`ifetch()`[63a]), 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()`[35a] function is the decoder: given a 32-bit instruction word, it returns an index into `itab`[22a]. 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` 35a⟩≡ (119c)

```
int
arm_class(instruction w)
{
    // between 0 and 7
    int class;
    // enum<opcode>
    int op;
    ⟨arm_class() locals 36a⟩

    class = (w >> 25) & 0x7;
    switch(class) {
    ⟨arm_class() class cases 36b⟩
    default:
        op = OUNDEF;
        break;
    }
    return op;
}
```

Uses `OUNDEF` 21a.

5.4 Arithmetic and logic

With the decode loop and `arm_class()`[35a] 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 × 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* 35b⟩≡ (21a) 36c▷

```
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* 36a⟩≡ (35a)
```
  int x;
```

⟨arm_class() *class cases* 36b⟩≡ (35a) 44a▷
```
  case 0: /* data processing r,r,r */
      x = ((w >> 4) & 0xf);
      ⟨arm_class() class 0, if x is 0x9 36d⟩
      ⟨arm_class() class 0, if x has 0x9 bits 48b⟩
      // else

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

      ⟨arm_class() class 0, adjust op for operands mode 36f⟩
      break;
```

⟨*arith/logic opcodes* 36c⟩+≡ (21a) ◁35b 36e▷
```
  OMUL    = 64,
  OMULA   = 65,
```

⟨arm_class() *class 0, if x is 0x9* 36d⟩≡ (36b)
```
  /* mul, swp, mull */
  if(x == 0x9) {
      op = CMUL;
      ⟨arm_class() class 0, when x == 0x9, if bit 24 set 48d⟩
      ⟨arm_class() class 0, when x == 0x9, if bit 23 set 45c⟩
      if(w & (1<<21))
          op++;  /* mla */
      break;
  }
```
Uses CMUL 36e.

⟨*arith/logic opcodes* 36e⟩+≡ (21a) ◁36c 36g▷
```
  CMUL    = 64,
```

⟨arm_class() *class 0, adjust op for operands mode* 36f⟩≡ (36b)
```
  if(w & (1<<4))
    op += CARITH2;
  else
   if((w & (31<<7)) || (w & (1<<5)))
    op += CARITH1;
  // else op += CARITH0
```
Uses CARITH1 36g and CARITH2 36g.

⟨*arith/logic opcodes* 36g⟩+≡ (21a) ◁36e 44b▷
```
  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`[37b] 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* 37a⟩+≡ (22a) ◁22b 38a▷
```
  [OMUL]    =  { Imul, "MUL", Iarith },
  [OMULA]   =  { Imula, "MULA", Iarith },
```
Uses `Iarith` 21b, `Imul()` 37b, and `Imula()` 37c.

⟨*function* `Imul` 37b⟩≡ (119c)
```
  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` 22e, `reg` 22d, and `undef()` 113d.

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()`[113d]), since it makes no sense to jump to an address that is the result of a multiplication. The general data-processing handlers (`Idp0`[38b] 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` 37c⟩≡ (119c)
```
  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` 22e, `reg` 22d, and `undef()` 113d.

5.4.3 `Idp()` and `dpex()`

The three `Idp` functions (`Idp0`[38b], `Idp1`[41c], `Idp2`[42b]) 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()`[39d] 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* 38a⟩+≡ (22a) ◁37a 41b▷

```
//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` 21b and `Idp0()` 38b.

⟨*function* `Idp0` 38b⟩≡ (119c)

```
/*
 * 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 39a⟩
    o2 = reg.r[rm];
    ⟨adjust o2 if rm is REGPC 39b⟩

    dpex(inst, o1, o2, rd);

    ⟨Idp0() trace ??⟩
    ⟨Idpx() compensate REGPC 39c⟩
}
```

Uses `dpex()` 39d and `reg` 22d.

Unlike `Imul`[37b], 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

[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.

(e.g., `MOV R14, R15` for a return), the handler subtracts 4 to compensate for the `REGPC += 4` at the end of `run()`[34c]:

⟨*adjust o1 if rn is REGPC* 39a⟩≡ (45a 42b 41c 38b)
```
if(rn == REGPC)
    o1 += 8;
```
Uses REGPC 22e.

⟨*adjust o2 if rm is REGPC* 39b⟩≡ (42b 41c 38b)
```
if(rm == REGPC)
    o2 += 8;
```
Uses REGPC 22e.

⟨`Idpx()` *compensate REGPC* 39c⟩≡ (45a 42b 41c 38b)
```
if(rd == REGPC)
    reg.r[rd] -= 4;
```
Uses REGPC 22e and reg 22d.

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` 39d⟩≡ (119c)
```
void
dpex(instruction inst, long o1, long o2, int rd)
{
    bool cbit = false;

    switch((inst>>21) & 0xf) {
    ⟨dpex() switch arith/logic opcode cases 39e⟩
    }
    ⟨dpex() if Sbit 40b⟩
}
```

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* 39e⟩≡ (39d) 39f▷
```
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 35b, OEOR 35b, OORR 35b, and reg 22d.

⟨`dpex()` *switch arith/logic opcode cases* 39f⟩+≡ (39d) ◁39e 40d▷
```
case OBIC:
    reg.r[rd] = o1 & ~o2;
    cbit = true;
    break;
```
Uses OBIC 35b and reg 22d.

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` 40a⟩≡ (27e)
```
Sbit  = 1<<20,
```

⟨`dpex()` *if Sbit* 40b⟩≡ (39d)
```
if(inst & Sbit) {
    if(cbit)
        reg.cbit = reg.cout;
    reg.cc1 = reg.r[rd];
    reg.cc2 = 0;
    reg.compare_op = CCcmp;
}
```
Uses `CCcmp` 56e, `Sbit` 40a, and `reg` 22d.

⟨`Registers` *other fields* 40c⟩+≡ (22c) ◁34b 56b▷
```
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* 40d⟩+≡ (39d) ◁39f 46b▷
```
case OADD:
    ⟨dpex() if calltree, when add operation 98d⟩
    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 56e, OADD 35b, Sbit 40a, and reg 22d.

⟨*macro* XCAST 41a⟩≡ (119c)

```
#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* 41b⟩+≡ (22a) ◁38a 42a▷

```
// 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 21b and Idp1() 41c.

⟨*function* Idp1 41c⟩≡ (119c)

```
/*
 * 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 39a⟩
    o2 = reg.r[rm];
```

```
    ⟨adjust o2 if rm is REGPC 39b⟩

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

    ⟨Idp1() trace ??⟩
    ⟨Idpx() compensate REGPC 39c⟩
}
```

Uses dpex() 39d, reg 22d, and shift() 43b.

```
⟨itab elements 42a⟩+≡                                    (22a) ◁41b 44c▷
  // 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 21b and Idp2() 42b.

```
⟨function Idp2 42b⟩≡                                     (119c)
  /*
   * 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 39a⟩
      o2 = reg.r[rm];
      ⟨adjust o2 if rm is REGPC 39b⟩
      o3 = reg.r[rs];
      ⟨adjust o3 if rs is REGPC 43a⟩

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

      ⟨Idp2() trace ??⟩
      ⟨Idpx() compensate REGPC 39c⟩
  }
```

Uses dpex() 39d, reg 22d, and shift() 43b.

⟨*adjust o3 if rs is REGPC* 43a⟩≡ (42b)
```
if(rs == REGPC)
    o3 += 8;
```
Uses REGPC 22e.

⟨*function* shift 43b⟩≡ (119c)
```
long
shift(long v, int st, int sc, bool isreg)
{
    ⟨shift() if sc is 0 43c⟩
    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 22d.

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* 43c⟩≡ (43b)
```
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` 22d.

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* 44a⟩+≡ (35a) ◁36b 47d▷

```
  case 1: /* data processing i,r,r */
   op = CARITH3 + ((w >> 21) & 0xf);
   break;
```

Uses `CARITH3` 44b.

⟨*arith/logic opcodes* 44b⟩+≡ (21a) ◁36g 45b▷

```
  CARITH3 = 48, // i,r,r
```

⟨`itab` *elements* 44c⟩+≡ (22a) ◁42a 45d▷

```
  //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` 21b and `Idp3()` 45a.

⟨*function* `Idp3` 45a⟩≡ (119c)

```
/*
 * 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 39a⟩

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

    dpex(inst, o1, o2, rd);

    ⟨Idp3() trace ??⟩
    ⟨Idpx() compensate REGPC 39c⟩
}
```

Uses `dpex()` 39d and `reg` 22d.

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* 45b⟩+≡ (21a) ◁44b

```
OMULLU  = 66,
OMULALU = 67,
OMULL   = 68,
OMULAL  = 69,
```

⟨`arm_class()` *class 0, when x == 0x9, if bit 23 set* 45c⟩≡ (36d)

```
if(w & (1<<23)) { /* mullu */
    op = CMUL+2;
    if(w & (1<<22)) /* mull */
        op = CMUL+4;
}
```

Uses `CMUL` 36e.

⟨`itab` *elements* 45d⟩+≡ (22a) ◁44c 48e▷

```
[OMULLU]  =  { Imull, "MULLU", Iarith },
[OMULALU] =  { Imull, "MULALU", Iarith },
[OMULL]   =  { Imull, "MULL",  Iarith },
[OMULAL]  =  { Imull, "MULAL", Iarith },
```

Uses `Iarith` 21b and `Imull()` 46a.

⟨*function* `Imull` 46a⟩≡ (119c)

```
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` 22e, `reg` 22d, and `undef()` 113d.

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* 46b⟩+≡ (39d) ◁40d 46c▷

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

Uses `CCcmp` 56e, `OCMN` 35b, `Sbit` 40a, and `reg` 22d.

⟨`dpex()` *switch arith/logic opcode cases* 46c⟩+≡ (39d) ◁46b 47a▷

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

```
    }
    return;
```

Uses CCcmp 56e, ORSB 35b, Sbit 40a, and reg 22d.

⟨dpex() *switch arith/logic opcode cases* 47a⟩+≡ (39d) ◁46c 47b▷

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

Uses OADC 35b, ORSC 35b, OSBC 35b, and undef() 113d.

⟨dpex() *switch arith/logic opcode cases* 47b⟩+≡ (39d) ◁47a 55d▷

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

Uses OMOV 35b, OMVN 35b, and reg 22d.

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* 47c⟩≡ (21a) 47f▷

```
OLDW    = 70,
OLDB    = 71,
OSTW    = 72,
OSTB    = 73,
```

⟨arm_class() *class cases* 47d⟩+≡ (35a) ◁44a 47e▷

```
case 2: /* load/store byte/word i(r) */
 //                          OxxB?           OSTx?
 op = CMEM_BASIS + CMEM0 + ((w >> 22) & 0x1) + ((w >> 19) & 0x2);
 break;
```

Uses CMEM0 47f and CMEM_BASIS 47f.

⟨arm_class() *class cases* 47e⟩+≡ (35a) ◁47d 53b▷

```
case 3: /* load/store byte/word (r)(r) */
 //                          OxxB?           OSTx?
 op = CMEM_BASIS + CMEM1 + ((w >> 22) & 0x1) + ((w >> 19) & 0x2);
 break;
```

Uses CMEM1 47f and CMEM_BASIS 47f.

⟨*memory opcodes* 47f⟩+≡ (21a) ◁47c 48a▷

```
CMEM_BASIS = 70,
CMEM0 = 0, // i(r)
CMEM1 = 4, // (r),(r)
CMEM2 = 8, // byte signed or half word
```

⟨*memory opcodes* 48a⟩+≡ (21a) ◁47f 48c▷
```
OLDH    = 78,
OLDBU   = 79,
OSTH    = 80,
OSTBU   = 81,
```

⟨`arm_class()` *class 0, if x has 0x9 bits* 48b⟩≡ (36b)
```
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` 47f and `CMEM_BASIS` 47f.

⟨*memory opcodes* 48c⟩+≡ (21a) ◁48a 53a▷
```
OSWPW = 82,
OSWPBU = 83,
```

⟨`arm_class()` *class 0, when x == 0x9, if bit 24 set* 48d⟩≡ (36d)
```
if(w & (1<<24)) {
    op = OSWPW;
    if(w & (1<<22))
         op = OSWPBU;
    break;
}
```
Uses `OSWPBU` 48c and `OSWPW` 48c.

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* 48e⟩+≡ (22a) ◁45d 49a▷
```
[OSWPW] =   { Iswap,  "SWPW", Imem },
[OSWPBU] =  { Iswap,  "SWPBU",Imem },
```
Uses `Imem` 21b and `Iswap()` 48f.

⟨*function* `Iswap` 48f⟩≡ (119c)
```
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() 64d, getmem_w() 65a, putmem_b() 66b, putmem_w() 67a, and reg 22d.

The getmem_b[64d]/getmem_w[65a] and putmem_b[66b]/putmem_w[67a] 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[50], 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* 49a⟩+≡ (22a) ◁48e 49b▷

```
// load/store w/ub i,r
[OLDW +CMEM0] =  { Imem1,  "MOVW", Imem },
[OLDB +CMEM0] =  { Imem1,  "MOVB", Imem },
[OSTW +CMEM0] =  { Imem1,  "MOVW", Imem },
[OSTB +CMEM0] =  { Imem1,  "MOVB", Imem },
```

Uses Imem 21b and Imem1() 50.

⟨itab *elements* 49b⟩+≡ (22a) ◁49a 51a▷

```
// 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 21b and Imem1() 50.

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` 50⟩≡ (119c)

```
/*
 * 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 22e, getmem_b() 64d, getmem_w() 65a, putmem_b() 66b, putmem_w() 67a, reg 22d, and shift() 43b.

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* 51a⟩+≡ (22a) ◁49b 53d▷

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

Uses Imem 21b and Imem2() 51b.

⟨*function* Imem2 51b⟩≡ (119c)

```
/*
 * 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 22e, getmem_b() 64d, getmem_h() 64e, putmem_b() 66b, putmem_h() 66c, and reg 22d.

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* 53a⟩+≡ (21a) ◁48c 53c▷
```
  OLDM = 84,
  OSTM = 85,
```

⟨`arm_class()` *class cases* 53b⟩+≡ (35a) ◁47e 54b▷
```
  case 4: /* block data transfer (r)(r) */
   op = CBLOC + ((w >> 20) & 0x1);
   break;
```
Uses `CBLOC` 53c.

⟨*memory opcodes* 53c⟩+≡ (21a) ◁53a
```
  CBLOC   = 84,
```

⟨`itab` *elements* 53d⟩+≡ (22a) ◁51a 54d▷
```
  // block move r,r
  [OLDM] =  { Ilsm,  "LDM", Imem },
  [OSTM] =  { Ilsm,  "STM", Imem },
```
Uses `Ilsm()` 53e and `Imem` 21b.

⟨*function* `Ilsm` 53e⟩≡ (119c)
```
  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() 65a, putmem_w() 67a, reg 22d, and undef() 113d.

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* 54a⟩≡ (21a) 54c▷

```
OB = 86,
OBL = 87,
```

⟨arm_class() *class cases* 54b⟩+≡ (35a) ◁53b 58c▷

```
case 5: /* branch / branch link */
 op = CBRANCH + ((w >> 24) & 0x1);
 break;
```

Uses CBRANCH 54c.

⟨*branching opcodes* 54c⟩+≡ (21a) ◁54a

```
CBRANCH = 86,
```

5.6.2 Simple branching

⟨itab *elements* 54d⟩+≡ (22a) ◁53d 55b▷

```
// branch
[OB]  =  { Ib, "B", Ibranch },
```

Uses Ib() 55a and Ibranch 21b.

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()`[34c]:

⟨*function* `Ib` 55a⟩≡ (119c)
```
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` 22e and `reg` 22d.

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* 55b⟩+≡ (22a) ◁54d 58d▷
```
[OBL] =  { Ibl, "BL", Ibranch },
```
Uses `Ibl()` 55c and `Ibranch` 21b.

⟨*function* `Ibl` 55c⟩≡ (119c)
```
void
Ibl(instruction inst)
{
    long v;
    Symbol s;

    v = inst & 0xffffff;
    v = reg.r[REGPC] + (v << 2) + 8;
    ⟨Ibl() trace ??⟩
    ⟨Ibl() if calltree 99a⟩
    reg.r[REGLINK] = reg.r[REGPC] + 4;
    reg.r[REGPC] = v - 4;
}
```
Uses `REGLINK` 22e, `REGPC` 22e, and `reg` 22d.

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* 55d⟩+≡ (39d) ◁47b 56a▷
```
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 56e, OCMP 35b, OSUB 35b, Sbit 40a, and reg 22d.

⟨dpex() *switch arith/logic opcode cases* 56a⟩+≡ (39d) ◁55d

```
  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 56e, CCtst 56e, OTEQ 35b, OTST 35b, Sbit 40a, and reg 22d.

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* 56b⟩+≡ (22c) ◁40c 56d▷

```
  // actually only 4 bits (16 possibilities)
  int instr_cond;
```

⟨run() *set reg.cond* 56c⟩≡ (34c)

```
  reg.instr_cond = (reg.instr>>28) & 0xf;
```

Uses reg 22d.

⟨Registers *other fields* 56d⟩+≡ (22c) ◁56b 57b▷

```
  // enum<compare_op>
  int compare_op;
```

⟨*enum* compare_op 56e⟩≡ (115)

```
  enum compare_op
  {
      CCcmp,
      CCtst,
      CCteq,
  };
```

⟨run() *switch* reg.compare_op *to set* execute 57a⟩≡ (34c)
```
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 56e, CCteq 56e, CCtst 56e, bout 25d, reg 22d, runcmp() 57c, runteq() 57e, and runtst() 58a.

⟨Registers *other fields* 57b⟩+≡ (22c) ◁56d
```
long cc1;
long cc2;
```

⟨*function* runcmp 57c⟩≡ (119c)
```
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 25d, reg 22d, and undef() 113d.

⟨*constant* SIGNBIT 57d⟩≡ (27e)
```
SIGNBIT  = 0x80000000,
```

⟨*function* runteq 57e⟩≡ (119c)
```
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 57d, bout 25d, reg 22d, and undef() 113d.

⟨*function* runtst 58a⟩≡ (119c)

```
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 57d, bout 25d, reg 22d, and undef() 113d.

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()[69] which emulates the system call by proxying it to the host OS (see Chapter 7).

⟨*syscall opcodes* 58b⟩≡ (21a)

```
OSWI = 88,
```

⟨arm_class() *class cases* 58c⟩+≡ (35a) ◁54b

```
case 7: /* coprocessor crap */ // and syscall
 if((w >> 25) & 0x1)
   op = OSWI;
 else
   op = OUNDEF; // coprocessor stuff not handled
 break;
```

Uses OSWI 58b and OUNDEF 21a.

⟨itab *elements* 58d⟩+≡ (22a) ◁55b

```
[OSWI] =  { Ssyscall,  "SWI", Isyscall },
```

Uses Isyscall 21b and Ssyscall() 69.

5.8 Unimplemented instructions

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` 60⟩≡ (119a)

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

    ⟨page_of_vaddr() TLB handling 62f⟩

    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 61a⟩
            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 27e, Nseg 23a, bout 25d, errjmp 113b, fatal() 113a, and memory 23c.

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* 61a⟩≡ (60) 61b▷

```
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 27e, Text 23a, emalloc() 114a, fatal() 113a, and text 25b.

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* 61b⟩+≡ (60) ◁61a 62a▷

```
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 27e, Data 23a, emalloc() 114a, fatal() 113a, and text 25b.

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* 62a⟩+≡ (60) ◁61b
```
case Bss:
case Stack:
    *p = emalloc(BY2PG);
    return *p;
```
Uses `BY2PG` 27e, `Bss` 23a, `Stack` 23a, and `emalloc()` 114a.

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` 62b⟩≡ (115)
```
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` 62c.

⟨*constant* `Nmaxtlb` 62c⟩≡ (115)
```
#define Nmaxtlb 64
```

⟨*global* `tlb` 62d⟩≡ (117b)
```
Tlb tlb;
```

⟨`main()` *tlb initialisation* 62e⟩≡ (25e)
```
tlb.on = true;
tlb.tlbsize = 24;
```
Uses `tlb` 62d.

⟨`page_of_vaddr()` *TLB handling* 62f⟩≡ (60)
```
if(tlb.on)
    dotlb(addr);
```
Uses `dotlb()` 62g and `tlb` 62d.

`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` 62g⟩≡ (119a)
```
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 27e and tlb 62d.

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.

⟨*function* ifetch 63a⟩≡ (119a)

```
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 64b⟩
    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 27e, PROFGRAN 103e, bout 25d, errjmp 113b, iprof 103d, page_of_vaddr() 60, and textbase 28d.

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 63b⟩≡ (115)

```
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` 64a⟩≡ (117b)
```
Icache icache;
```

⟨`ifetch()` *instruction cache handling* 64b⟩≡ (63a)
```
if(icache.on)
    updateicache(addr);
```
Uses `icache` 64a and `updateicache()` 64c.

⟨*function* `updateicache` 64c⟩≡ (117a)
```
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` 64d⟩≡ (119a)
```
byte
getmem_b(uintptr addr)
{
    byte *va;

    ⟨getmem_x() if membpt 102c⟩

    va = page_of_vaddr(addr);
    va += addr&(BY2PG-1);
    return va[0];
}
```
Uses `BY2PG` 27e and `page_of_vaddr()` 60.

⟨*function* `getmem_h` 64e⟩≡ (119a)
```
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 102c⟩

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

    return va[1]<<8 | va[0];
}
```

Uses BY2PG 27e, getmem_h() 64e, and page_of_vaddr() 60.

⟨*function* getmem_w 65a⟩≡ (119a)

```
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 102c⟩

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

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

Uses BY2PG 27e, getmem_w() 65a, and page_of_vaddr() 60.

⟨*function* getmem_v 65b⟩≡ (119a)

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

Uses getmem_w() 65a.

`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 65c⟩≡ (119a)

```
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() 64d.

⟨*function* getmem_4 66a⟩≡ (119a)

```
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() 64d.

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 66b⟩≡ (119a)

```
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 102d⟩
}
```

Uses BY2PG 27e and page_of_vaddr() 60.

⟨*function* putmem_h 66c⟩≡ (119a)

```
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 102d⟩
}
```

Uses BY2PG 27e, bout 25d, errjmp 113b, and page_of_vaddr() 60.

⟨*function* putmem_w 67a⟩≡ (119a)

```
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 102d⟩
}
```

Uses BY2PG 27e, bout 25d, errjmp 113b, and page_of_vaddr() 60.

⟨*function* putmem_v 67b⟩≡ (119a)

```
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() 67a.

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` 68⟩≡ (122b)

```
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() 82f, sysawait() 80e, sysbind() 78a, sysbrk() 71d, syschdir() 77b, sysclose() 73a, syscreate() 76a, sysdup() 78b, syserrstr() 71c, sysexec() 80d, sysexits() 79a, sysfauth() 83a, sysfd2path() 77a, sysfstat() 75c, sysfversion() 83b, sysfwstat() 81b, sysmount() 82d, sysnop() 71a, sysnoted() 81c, sysnotify() 80b, sysopen() 72, syspipe() 79c, syspread() 74a, syspwrite() 75a, sysremove() 76b, sysrendezvous() 82c, sysrfork() 80c, sysseek() 74b, syssegattach() 81d, syssegbrk() 82b, syssegdetach() 81e, syssegflush() 82a, syssegfree() 81f, syssleep() 79b, sysstat() 75b, sysunmount() 82e, and syswstat() 81a.

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.

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 69⟩≡ (122b)

```
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 22e, bout 25d, dumpreg() 96a, itrace() 112b, reg 22d, sysctab, systab 68, and trace 112a.

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 70a⟩≡ (115)

```
// for memio()
enum
{
    MemRead,
    MemReadstring,
    MemWrite,
};
```

⟨*function* memio 70b⟩≡ (119a)

```
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 70a, MemReadstring 70a, MemWrite 70a, bout 25d, emalloc() 114a, errjmp 113b, fatal() 113a, getmem_b() 64d, and putmem_b() 66b.

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 71a⟩≡ (122b)

```
void
sysnop(void)
{
    Bprint(bout, "nop system call %s\n", sysctab[reg.r[1]]);
    ⟨sysnop strace 96e⟩
}
```

Uses bout 25d, reg 22d, and sysctab.

⟨*global* errbuf 71b⟩≡ (122b)

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

⟨*function* syserrstr 71c⟩≡ (122b)

```
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 70a, REGRET 22e, REGSP 22e, errbuf 71b, getmem_w() 65a, itrace() 112b, memio() 70b, reg 22d, and sysdbg 96d.

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 71d⟩≡ (122b)

```
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 27e, Bss 23a, Data 23a, REGRET 22e, REGSP 22e, Stack 23a, datasize 29b, erealloc() 114b, errbuf 71b, getmem_w() 65a, itrace() 112b, memory 23c, reg 22d, and sysdbg 96d.

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 72⟩≡ (122b)

```
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 70a, REGRET 22e, REGSP 22e, errbuf 71b, getmem_w() 65a, itrace() 112b, memio() 70b, reg 22d, and sysdbg 96d.

⟨*function* sysclose 73a⟩≡ (122b)
```
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 22e, REGSP 22e, errbuf 71b, getmem_w() 65a, itrace() 112b, reg 22d, and sysdbg 96d.

⟨*function* sysread 73b⟩≡ (122b)
```
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 70a, REGRET 22e, REGSP 22e, bin 25d, emalloc() 114a, errbuf 71b, getmem_w() 65a, itrace() 112b, memio() 70b, reg 22d, and sysdbg 96d.

⟨*function* syspread 74a⟩≡ (122b)

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

Uses REGSP 22e, getmem_v() 65b, reg 22d, and sysread() 73b.

⟨*function* sysseek 74b⟩≡ (122b)

```
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 22e, errbuf 71b, getmem_v() 65b, getmem_w() 65a, itrace() 112b, putmem_v() 67b, reg 22d, and sysdbg 96d.

⟨*function* syswrite 74c⟩≡ (122b)

```
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 70a, REGRET 22e, REGSP 22e, bout 25d, errbuf 71b, getmem_w() 65a, itrace() 112b, memio() 70b, reg 22d, and sysdbg 96d.

⟨*function* syspwrite 75a⟩≡ (122b)

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

Uses REGSP 22e, getmem_v() 65b, reg 22d, and syswrite() 74c.

⟨*function* sysstat 75b⟩≡ (122b)

```
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 70a, MemWrite 70a, REGRET 22e, REGSP 22e, errbuf 71b, getmem_w() 65a, itrace() 112b, memio() 70b, reg 22d, and sysdbg 96d.

⟨*function* sysfstat 75c⟩≡ (122b)

```
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 70a, REGRET 22e, REGSP 22e, errbuf 71b, getmem_w() 65a, itrace() 112b, memio() 70b, reg 22d, and sysdbg 96d.

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 76a⟩≡ (122b)

```
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 70a, REGRET 22e, REGSP 22e, errbuf 71b, getmem_w() 65a, itrace() 112b, memio() 70b, reg 22d, and sysdbg 96d.

⟨*function* sysremove 76b⟩≡ (122b)

```
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 70a, REGRET 22e, REGSP 22e, errbuf 71b, getmem_w() 65a, itrace() 112b, memio() 70b, reg 22d, and sysdbg 96d.

⟨*function* sysfd2path 77a⟩≡ (122b)

```
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 70a, REGRET 22e, REGSP 22e, errbuf 71b, getmem_w() 65a, itrace() 112b, memio() 70b, reg 22d, and sysdbg 96d.

⟨*function* syschdir 77b⟩≡ (122b)

```
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 70a, REGRET 22e, REGSP 22e, errbuf 71b, getmem_w() 65a, itrace() 112b, memio() 70b, reg 22d, and sysdbg 96d.

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` 78a⟩≡ (122b)

```
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` 70a, `REGRET` 22e, `REGSP` 22e, `errbuf` 71b, `getmem_w()` 65a, `itrace()` 112b, `memio()` 70b, `reg` 22d, and `sysdbg` 96d.

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` 78b⟩≡ (122b)

```
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` 22e, `REGSP` 22e, `errbuf` 71b, `getmem_w()` 65a, `itrace()` 112b, `reg` 22d, and `sysdbg` 96d.

⟨*constant* `OERRLEN` 78c⟩≡ (122b)

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

⟨*function* sysexits 79a⟩≡ (122b)

```
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 70a, OERRLEN-8 78c, REGSP 22e, bout 25d, count 34a, getmem_w() 65a, itrace() 112b, memio() 70b, reg 22d, and sysdbg 96d.

⟨*function* syssleep 79b⟩≡ (122b)

```
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 22e, REGSP 22e, errbuf 71b, getmem_w() 65a, itrace() 112b, reg 22d, and sysdbg 96d.

⟨*function* syspipe 79c⟩≡ (122b)

```
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 22e, REGSP 22e, errbuf 71b, getmem_w() 65a, itrace() 112b, putmem_w() 67a, reg 22d, and sysdbg 96d.

⟨*global* nofunc 80a⟩≡ (122b)

```
ulong nofunc;
```

⟨*function* sysnotify 80b⟩≡ (122b)

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

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

Uses REGRET 22e, REGSP 22e, getmem_w() 65a, itrace() 112b, nofunc 80a, reg 22d, and sysdbg 96d.

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 80c⟩≡ (122b)

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

Uses REGARG 22e, bout 25d, reg 22d, and sysctab.

⟨*function* sysexec 80d⟩≡ (122b)

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

Uses REGARG 22e, bout 25d, reg 22d, and sysctab.

⟨*function* sysawait 80e⟩≡ (122b)

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

Uses REGARG 22e, bout 25d, reg 22d, and sysctab.

⟨*function* `syswstat` 81a⟩≡ (122b)

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

Uses `REGARG` 22e, `bout` 25d, `reg` 22d, and `sysctab`.

⟨*function* `sysfwstat` 81b⟩≡ (122b)

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

Uses `REGARG` 22e, `bout` 25d, `reg` 22d, and `sysctab`.

⟨*function* `sysnoted` 81c⟩≡ (122b)

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

Uses `REGARG` 22e, `bout` 25d, `reg` 22d, and `sysctab`.

⟨*function* `syssegattach` 81d⟩≡ (122b)

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

Uses `REGARG` 22e, `bout` 25d, `reg` 22d, and `sysctab`.

⟨*function* `syssegdetach` 81e⟩≡ (122b)

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

Uses `REGARG` 22e, `bout` 25d, `reg` 22d, and `sysctab`.

⟨*function* `syssegfree` 81f⟩≡ (122b)

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

Uses `REGARG` 22e, `bout` 25d, `reg` 22d, and `sysctab`.

⟨*function* `syssegflush` 82a⟩≡ (122b)

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

Uses `REGARG` 22e, `bout` 25d, `reg` 22d, and `sysctab`.

⟨*function* `syssegbrk` 82b⟩≡ (122b)

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

Uses `REGARG` 22e, `bout` 25d, `reg` 22d, and `sysctab`.

⟨*function* `sysrendezvous` 82c⟩≡ (122b)

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

Uses `REGARG` 22e, `bout` 25d, `reg` 22d, and `sysctab`.

⟨*function* `sysmount` 82d⟩≡ (122b)

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

Uses `REGARG` 22e, `bout` 25d, `reg` 22d, and `sysctab`.

⟨*function* `sysunmount` 82e⟩≡ (122b)

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

Uses `REGARG` 22e, `bout` 25d, `reg` 22d, and `sysctab`.

⟨*function* `sysalarm` 82f⟩≡ (122b)

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

Uses `REGARG` 22e, `bout` 25d, `reg` 22d, and `sysctab`.

⟨*function* `sysfauth` 83a⟩≡ (122b)

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

Uses `REGARG` 22e, `bout` 25d, `reg` 22d, and `sysctab`.

⟨*function* `sysfversion` 83b⟩≡ (122b)

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

Uses `REGARG` 22e, `bout` 25d, `reg` 22d, 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* 84a⟩+≡ (31c) ◁31b 84b▷
```
static char *cmdlet = ":$?/=>"; //$
```

⟨`cmd()` *locals* 84b⟩+≡ (31c) ◁84a 84c▷
```
char buf[128];
char addr[128];
```

⟨`cmd()` *locals* 84c⟩+≡ (31c) ◁84b 84d▷
```
char lastcmd[128];
```

⟨`cmd()` *locals* 84d⟩+≡ (31c) ◁84c
```
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* 84e⟩≡ (31c)
```
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` 25d, `cmdcount` 104a, and `nextc()` 85a.

⟨*function* `nextc` 85a⟩≡ (122a)

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

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

        return p;
    }
```

⟨`cmd()` *command cases* 85b⟩+≡ (31c) ◁31d 85c▷

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

Uses `dollar()` 88b.

⟨`cmd()` *command cases* 85c⟩+≡ (31c) ◁85b 86a▷

```
    case '/':
    case '?':
        dot = expr(addr);
        for(i = 0; i < cmdcount; i++)
            quesie(p+1);
        break;
```

Uses `cmdcount` 104a, `dot` 31a, and `expr()` 87b.

⟨cmd() *command cases* 86a⟩+≡ (31c) ◁85c 86b▷
```
case '=':
    eval(addr, p+1);
    break;
```
Uses eval() 94a.

⟨cmd() *command cases* 86b⟩+≡ (31c) ◁86a
```
case '>':
    setreg(addr, p+1);
    break;
```
Uses setreg() 95.

8.2 Interface

⟨*global* fmt 86c⟩≡ (122a)
```
char fmt = 'X';
```
Uses fmt 86c.

⟨*global* width 86d⟩≡ (122a)
```
int width = 60;
```
Uses width 86d.

⟨*global* inc 86e⟩≡ (122a)
```
int inc;
```

⟨*function* reset 86f⟩≡ (122a)
```
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 27e, Nseg 23a, bplist 99d, iprof 103d, memory 23c, and reg 22d.

⟨*function* numsym 87a⟩≡ (122a)

```
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 31a.

⟨*function* expr 87b⟩≡ (122a)

```
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` 25d, `dot` 31a, `nextc()` 85a, and `numsym()` 87a.

⟨*function* `buildargv` 88a⟩≡ (122a)

```
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` 88b⟩≡ (122a)

```
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 98c⟩
    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 25d, calltree 98b, dobplist() 100b, dumpdreg() 96c, dumpfreg() 96b, dumpreg() 96a, iprofile() 106g, isum() 104c, nextc() 85a, segsum() 106c, stktrace() 98a, sysdbg 96d, tlbsum() 105, and trace 112a.

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* 90a⟩≡ (32a) 91b▷

```
char tbuf[512];
```

⟨colon() *print current instruction* 90b⟩≡ (32a)

```
symoff(tbuf, sizeof(tbuf), dot, CTEXT);
Bprint(bout, tbuf);
if(fmt == 'z')
    printsource(dot);
```

Uses bout 25d, dot 31a, fmt 86c, and printsource() 97b.

⟨colon() *command which return cases* 90c⟩≡ (32a) 91a▷

```
case 'b':
    breakpoint(addr, cp+1);
    return;
```

Uses breakpoint() 100c.

⟨colon() *command which return cases* 91a⟩+≡ (32a) ◁90c
```
  case 'd':
      delbpt(addr);
      return;
```
Uses delbpt() 101a.

⟨colon() *locals* 91b⟩+≡ (32a) ◁90a
```
  int argc;
  char *argv[100];
```

⟨colon() *command cases* 91c⟩+≡ (32a) ◁32b 91d▷
```
  case 'r':
      reset();
      argc = buildargv(cp+1, argv, 100);
      initstk(argc, argv);
      count = 0;
      atbpt = false;
      run();
      break;
```
Uses atbpt 100a, buildargv() 88a, count 34a, initstk() 30, reset() 86f, and run() 34c.

⟨colon() *command cases* 91d⟩+≡ (32a) ◁91c
```
  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 100a, count 34a, nextc() 85a, and run() 34c.

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 91e⟩≡ (122a)
```
  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 25d, dot 31a, getmem_2() 65c, getmem_4() 66a, getmem_b() 64d, inc 86e, printsource() 97b, symmap 27a, and width 86d.

⟨*function* eval 94a⟩≡ (122a)

```
    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 25d, expr() 87b, fmt 86c, nextc() 85a, and pfmt() 91e.

⟨*function* quesie 94b⟩≡ (122a)

```
    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` 95⟩≡ (122a)

```
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 22e, REGSP 22e, bout 25d, dot 31a, expr() 87b, nextc() 85a, and reg 22d.

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 96a⟩≡ (118a)

```
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 22e, REGSP 22e, bout 25d, and reg 22d.

⟨*function* dumpfreg 96b⟩≡ (118a)

```
void
dumpfreg(void)
{
}
```

⟨*function* dumpdreg 96c⟩≡ (118a)

```
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 96d⟩≡ (117b)

```
bool sysdbg;
```

⟨sysnop *strace* 96e⟩≡ (71a)

```
if(sysdbg)
    itrace("nop()");
```

Uses itrace() 112b and sysdbg 96d.

8.5.2 Stack trace

⟨*constant* STRINGSZ 97a⟩≡ (119b)
```
#define STRINGSZ 128
```

⟨*function* printsource 97b⟩≡ (119b)
```
/*
 * 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 97a and bout 25d.

⟨*function* printlocals 97c⟩≡ (119b)
```
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 25d and getmem_4() 66a.

⟨*function* printparams 97d⟩≡ (119b)
```
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 25d and getmem_4() 66a.

⟨*constant* STARTSYM 97e⟩≡ (119b)
```
#define STARTSYM "_main"
```

⟨*constant* FRAMENAME 97f⟩≡ (119b)
```
#define FRAMENAME ".frame"
```

⟨*function* stktrace 98a⟩≡ (119b)

```
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 97f, REGLINK 22e, REGPC 22e, REGSP 22e, STARTSYM-6 97e, bout 25d, getmem_4() 66a, printlocals() 97c, printparams() 97d, printsource() 97b, and reg 22d.

8.5.3 Call tree trace

⟨*global* calltree 98b⟩≡ (117b)

```
bool calltree;
```

⟨dollar() *t cases* 98c⟩≡ (88b)

```
case 'c':
    calltree = true;
    break;
```

Uses calltree 98b.

⟨dpex() *if calltree, when add operation* 98d⟩≡ (40d)

```
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 22e, REGRET 22e, bout 25d, calltree 98b, and reg 22d.

⟨Ibl() *if calltree* 99a⟩≡ (55c)

```
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 22e, REGSP 22e, bout 25d, calltree 98b, printparams() 97d, printsource() 97b, and reg 22d.

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).

⟨*enum* breakpoint_kind 99b⟩≡ (115)

```
enum breakpoint_kind
{
    Instruction  = 1,

    Read  = 2,
    Write  = 4,
    Access  = Read|Write,

    Equal  = 4|8,
};
```

Uses Read 99b and Write 99b.

⟨*struct* Breakpoint 99c⟩≡ (115)

```
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 99e⟩
};
```

⟨*global* bplist 99d⟩≡ (117b)

```
// list<Breakpoint> (next = Breakpoint.next)
Breakpoint *bplist;
```

⟨Breakpoint *extra fields* 99e⟩≡ (99c)

```
Breakpoint* next;  /* Link to next one */
```

⟨*global* atbpt 100a⟩≡ (117b)
```
bool atbpt;
```

⟨*function* dobplist 100b⟩≡ (118b)
```
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 99b, Equal 99b, Instruction 99b, Read 99b, Write 99b, bout 25d, and bplist 99d.

⟨*function* breakpoint 100c⟩≡ (118b)
```
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 99b, Equal 99b, Instruction 99b, Read 99b, Write 99b, bplist 99d, cmdcount 104a, emalloc() 114a, expr() 87b, membpt 102b, and nextc() 85a.

⟨*function* delbpt 101a⟩≡ (118b)

```
    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 99b, bout 25d, bplist 99d, expr() 87b, and membpt 102b.

⟨*function* brkchk 101b⟩≡ (118b)

```
    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 99b, atbpt 100a, bplist 99d, count 34a, and getmem_4() 66a.

8.6.1 Code breakpoint

⟨run() *check for breakpoints* 102a⟩≡ (34c)

```
if(bplist)
    brkchk(reg.r[REGPC], Instruction);
```

Uses Instruction 99b, REGPC 22e, bplist 99d, brkchk() 101b, and reg 22d.

8.6.2 Memory breakpoint

⟨*global* membpt 102b⟩≡ (117b)

```
bool membpt;
```

⟨getmem_x() *if membpt* 102c⟩≡ (65a 64)

```
if(membpt)
    brkchk(addr, Read);
```

Uses Read 99b, brkchk() 101b, and membpt 102b.

⟨putmem_x() *if membpt* 102d⟩≡ (67a 66)

```
if(membpt)
    brkchk(addr, Write);
```

Uses Write 99b, brkchk() 101b, and membpt 102b.

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.

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* 103a⟩≡ (21c)
```
// profiling info
int count;
int taken;
int useddelay;
```

⟨`run()` *profile current instruction class* 103b⟩≡ (34c)
```
// profiling
reg.ip->count++;
```
Uses `reg` 22d.

⟨*function* `Percent` 103c⟩≡ (121b)
```
#define Percent(num, max) ((max)?((num)*100)/(max):0)
```

⟨*global* `iprof` 103d⟩≡ (117b)
```
ulong *iprof;
```

⟨*constant* `PROFGRAN` 103e⟩≡ (27e)
```
PROFGRAN = 4,
```

⟨`initmemory()` *Text segment initilisation* 103f⟩+≡ (28a) ◁28e
⟨`initmemory()` *iprof allocation* 103g⟩

⟨`initmemory()` *iprof allocation* 103g⟩≡ (103f)
```
iprof = emalloc(((s->end - s->base)/PROFGRAN)*sizeof(long));
```
Uses `PROFGRAN` 103e, `emalloc()` 114a, and `iprof` 103d.

⟨*global* `tables` 103h⟩≡ (121b)
```
Inst *tables[] = { itab, 0 };
```
Uses `itab` 22a.

⟨*global* cmdcount 104a⟩≡ (117b)
```
int cmdcount;
```

⟨*global* nopcount 104b⟩≡ (117b)
```
int nopcount;
```

`isum()`[104c] 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 104c⟩≡ (121b)
```
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 21b, Ibranch 21b, Imem 21b, Isyscall 21b, Percent-1 103c, bout 25d, fatal() 113a, itab 22a, nopcount 104b, and tables 103h.

⟨*function* tlbsum 105⟩≡ (121b)

```
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 103c, bout 25d, and tlb 62d.

⟨*global* stype 106a⟩≡ (121b)
```
char *stype[] = { "Stack", "Text", "Data", "Bss" };
```

⟨Segment *profiling fields* 106b⟩≡ (23d)
```
int rss;
int refs;
```

⟨*function* segsum 106c⟩≡ (121b)
```
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 27e, Nseg 23a, bout 25d, memory 23c, and stype 106a.

⟨*struct* Prof 106d⟩≡ (121b)
```
struct Prof
{
    Symbol s;
    long count;
};
```

⟨*global* aprof 106e⟩≡ (121b)
```
// can't use prof, conflict with libc.h prof()
Prof aprof[5000];
```

⟨*function* profcmp 106f⟩≡ (121b)
```
int
profcmp(void *va, void *vb)
{
    Prof *a, *b;

    a = va;
    b = vb;
    return b->count - a->count;
}
```

`iprofile()`[106g] 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 106g⟩≡ (121b)
```
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 103e, aprof 106e, bout 25d, iprof 103d, printsource() 97b, profcmp() 106f, and textbase 28d.

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* 108⟩≡ (31c) 113c▷
```
  notify(catcher);
```
Uses `catcher()` 109.

⟨*function* `catcher` 109⟩≡ (122a)

```
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` 34a.

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 [Pad16b]: `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

⟨*global* `trace` 112a⟩≡ (117b)
```
bool trace;
```

⟨*function* `itrace` 112b⟩≡ (118a)
```
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` 25d and `reg` 22d.

⟨*global* `shtype` 112c⟩≡ (119c)
```
static char* shtype[4] =
{
    "<<",
    ">>",
    "->",
    "@>",
};
```

⟨*global* `cond` 112d⟩≡ (119c)
```
static char* cond[16] =
{
    ".EQ", ".NE", ".HS", ".LO",
    ".MI", ".PL", ".VS", ".VC",
    ".HI", ".LS", ".GE", ".LT",
    ".GT", ".LE", "", ".NO",
};
```

Appendix B

Error Management

⟨*function* `fatal` 113a⟩≡ (118a)

```
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` 113b⟩≡ (117b)

```
jmp_buf errjmp;
```

⟨`cmd()` *initialisation* 113c⟩+≡ (31c) ◁108

```
setjmp(errjmp);
```

Uses `errjmp` 113b.

⟨*function* `undef` 113d⟩≡ (119c)

```
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` 22e, `bout` 25d, `errjmp` 113b, and `reg` 22d.

Appendix C

Utilities

C.1 Memory Management

⟨*function* `emalloc` 114a⟩≡ (118a)

```
void *
emalloc(ulong size)
{
    void *a;

    a = malloc(size);
    if(a == nil)
        fatal(false, "no memory");

    memset(a, 0, size); //!!
    return a;
}
```

Uses `fatal()` 113a.

⟨*function* `erealloc` 114b⟩≡ (118a)

```
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()` 113a.

Appendix D

Extra Code

D.1 machine/5i/

D.1.1 machine/5i/arm.h

⟨machine/5i/arm.h 115⟩≡

```
/*
 * 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 20⟩

⟨enum breakpoint_kind 99b⟩

⟨struct Breakpoint 99c⟩

⟨enum ixxx 21b⟩

// added by pad
⟨enum opcode 21a⟩

⟨constant Nmaxtlb 62c⟩

⟨enum regxxx 22e⟩

⟨struct Tlb 62b⟩

⟨struct Icache 63b⟩


⟨struct Inst 21c⟩

⟨struct Registers 22c⟩

⟨enum memxxx 70a⟩
```

⟨*enum* `compare_op` 56e⟩

⟨*enum* `segment_kind` 23a⟩

⟨*struct* `Segment` 23d⟩

⟨*struct* `Memory` 23b⟩

```
// 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 27e⟩

Uses Breakpoint 99c, Icache 63b, Inst 21c, Memory 23b, Registers 22c, Segment 23d, and Tlb 62b.

D.1.2 machine/5i/icache.c

⟨machine/5i/icache.c 117a⟩≡

```
⟨basic includes 17⟩

⟨function updateicache 64c⟩
```

D.1.3 machine/5i/globals.c

⟨machine/5i/globals.c 117b⟩≡

```
⟨basic includes 17⟩

#include <tos.h>

//in run.c
//Inst itab[];
```

⟨*global* `reg` 22d⟩
⟨*global* `memory` 23c⟩
⟨*global* `text` 25b⟩
⟨*global* `trace` 112a⟩
⟨*global* `sysdbg` 96d⟩
⟨*global* `calltree` 98b⟩
⟨*global* `icache` 64a⟩
⟨*global* `tlb` 62d⟩
⟨*global* `count` 34a⟩
⟨*global* `errjmp` 113b⟩
⟨*global* `bplist` 99d⟩
⟨*global* `atbpt` 100a⟩
⟨*global* `membpt` 102b⟩
⟨*global* `cmdcount` 104a⟩
⟨*global* `nopcount` 104b⟩
⟨*global* `dot` 31a⟩
⟨*global* `bixxx` 25d⟩
⟨*global* `iprof` 103d⟩
⟨*global* `symmap` 27a⟩
⟨*global* `datasize` 29b⟩
⟨*global* `textbase` 28d⟩

D.1.4 `machine/5i/utils.c`

⟨`machine/5i/utils.c` 118a⟩≡
⟨*basic includes* 17⟩

⟨*function* `fatal` 113a⟩

⟨*function* `itrace` 112b⟩

⟨*function* `dumpreg` 96a⟩

⟨*function* `dumpfreg` 96b⟩

⟨*function* `dumpdreg` 96c⟩

⟨*function* `emalloc` 114a⟩

⟨*function* `erealloc` 114b⟩

D.1.5 `machine/5i/bpt.c`

⟨`machine/5i/bpt.c` 118b⟩≡
⟨*basic includes* 17⟩

```
#include <ctype.h>
```

⟨*function* `dobplist` 100b⟩

⟨*function* `breakpoint` 100c⟩

⟨*function* `delbpt` 101a⟩

⟨*function* `brkchk` 101b⟩

D.1.6 machine/5i/mem.c

⟨machine/5i/mem.c 119a⟩≡
⟨*basic includes* 17⟩

```
void*  page_of_vaddr(ulong);
```

⟨*function* ifetch 63a⟩

⟨*function* getmem_4 66a⟩

⟨*function* getmem_2 65c⟩

⟨*function* getmem_w 65a⟩

⟨*function* getmem_h 64e⟩

⟨*function* getmem_b 64d⟩

⟨*function* getmem_v 65b⟩

⟨*function* putmem_h 66c⟩

⟨*function* putmem_w 67a⟩

⟨*function* putmem_b 66b⟩

⟨*function* putmem_v 67b⟩

⟨*function* memio 70b⟩

⟨*function* dotlb 62g⟩

⟨*function* page_of_vaddr 60⟩

D.1.7 machine/5i/symbols.c

⟨machine/5i/symbols.c 119b⟩≡
⟨*basic includes* 17⟩

⟨*constant* STRINGSZ 97a⟩

⟨*function* printsource 97b⟩

⟨*function* printlocals 97c⟩

⟨*function* printparams 97d⟩

⟨*constant* STARTSYM 97e⟩
⟨*constant* FRAMENAME 97f⟩

⟨*function* stktrace 98a⟩

D.1.8 machine/5i/run.c

⟨machine/5i/run.c 119c⟩≡
⟨*basic includes* 17⟩

⟨*macro* XCAST 41a⟩

```
// 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 112c⟩
⟨*global* cond 112d⟩

⟨*global* itab 22a⟩

⟨*function* runcmp 57c⟩

⟨*function* runteq 57e⟩

⟨*function* runtst 58a⟩

⟨*function* run 34c⟩

⟨*function* undef 113d⟩

⟨*function* arm_class 35a⟩

⟨*function* shift 43b⟩

⟨*function* dpex 39d⟩

⟨*function* Idp0 38b⟩

⟨*function* Idp1 41c⟩

⟨*function* Idp2 42b⟩

⟨*function* Idp3 45a⟩

⟨*function* Imul 37b⟩

⟨*function* Imull 46a⟩

⟨*function* Imula 37c⟩

⟨*function* Iswap 48f⟩

⟨*function* Imem1 50⟩

⟨*function* Imem2 51b⟩

⟨*function* Ilsm 53e⟩

⟨*function* Ib 55a⟩

⟨*function* Ibl 55c⟩

D.1.9 machine/5i/5i.c

⟨machine/5i/5i.c 121a⟩≡
⟨*basic includes* 17⟩

```
#include <tos.h>
```

⟨*global* file 25a⟩
⟨*global* bxxx 25c⟩
⟨*global* fhdr 26a⟩

⟨*function* initmemory 28a⟩

⟨*function* inithdr 26b⟩

⟨*function* initstk 30⟩

⟨*function* main 25e⟩

D.1.10 machine/5i/stats.c

⟨machine/5i/stats.c 121b⟩≡
⟨*basic includes* 17⟩

⟨*function* Percent 103c⟩

```
typedef struct Prof Prof;
```

⟨*global* tables 103h⟩

⟨*function* isum 104c⟩

⟨*function* tlbsum 105⟩

⟨*global* stype 106a⟩

⟨*function* segsum 106c⟩

⟨*struct* Prof 106d⟩
⟨*global* aprof 106e⟩

⟨*function* profcmp 106f⟩

⟨*function* iprofile 106g⟩

Uses Prof 106d.

D.1.11 machine/5i/cmd.c

⟨machine/5i/cmd.c 122a⟩≡
⟨*basic includes* 17⟩

```
#include <ctype.h>
```

⟨*global* fmt 86c⟩
⟨*global* width 86d⟩
⟨*global* inc 86e⟩

⟨*function* reset 86f⟩

⟨*function* nextc 85a⟩

⟨*function* numsym 87a⟩

⟨*function* expr 87b⟩

⟨*function* buildargv 88a⟩

⟨*function* colon 32a⟩

⟨*function* dollar 88b⟩

⟨*function* pfmt 91e⟩

⟨*function* eval 94a⟩

⟨*function* quesie 94b⟩

⟨*function* catcher 109⟩

⟨*function* setreg 95⟩

⟨*function* cmd 31c⟩

D.1.12 machine/5i/syscall.c

⟨machine/5i/syscall.c 122b⟩≡
⟨*basic includes* 17⟩

```
//#define ODIRLEN 116 /* compatibility; used in _stat etc. */
```
⟨*constant* OERRLEN 78c⟩

⟨*global* errbuf 71b⟩
⟨*global* nofunc 80a⟩

```
#include "../../lib_core/libc/9syscall/sys.h"
```

⟨*global* sysctab ??⟩

⟨*function* sysnop 71a⟩

⟨*function* syserrstr 71c⟩
⟨*function* sysbind 78a⟩

⟨*function* `sysfd2path` 77a⟩

⟨*function* `syschdir` 77b⟩

⟨*function* `sysclose` 73a⟩

⟨*function* `sysdup` 78b⟩

⟨*function* `sysexits` 79a⟩

⟨*function* `sysopen` 72⟩

⟨*function* `sysread` 73b⟩

⟨*function* `syspread` 74a⟩

⟨*function* `sysseek` 74b⟩

⟨*function* `syssleep` 79b⟩

⟨*function* `sysstat` 75b⟩

⟨*function* `sysfstat` 75c⟩

⟨*function* `syswrite` 74c⟩

⟨*function* `syspwrite` 75a⟩

⟨*function* `syspipe` 79c⟩

⟨*function* `syscreate` 76a⟩

⟨*function* `sysbrk` 71d⟩

⟨*function* `sysremove` 76b⟩

⟨*function* `sysnotify` 80b⟩

⟨*function* `sysawait` 80e⟩
⟨*function* `sysrfork` 80c⟩
⟨*function* `syswstat` 81a⟩
⟨*function* `sysfwstat` 81b⟩
⟨*function* `sysnoted` 81c⟩
⟨*function* `syssegattach` 81d⟩
⟨*function* `syssegdetach` 81e⟩
⟨*function* `syssegfree` 81f⟩
⟨*function* `syssegflush` 82a⟩
⟨*function* `sysrendezvous` 82c⟩
⟨*function* `sysunmount` 82e⟩

⟨*function* `syssegbrk` 82b⟩
⟨*function* `sysmount` 82d⟩
⟨*function* `sysalarm` 82f⟩
⟨*function* `sysexec` 80d⟩

⟨*function* `sysfauth` 83a⟩

⟨*function* `sysfversion` 83b⟩

⟨*global* `systab` 68⟩

⟨*function* `Ssyscall` 69⟩

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.

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, 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) 13, 27, 111

[Pad15a] Yoann Padioleau. *Principia Softwarica: The ARM Assembler 5a*. 2015. cited page(s) 6, 15, 27, 50, 52, 110

[Pad15b] Yoann Padioleau. *Principia Softwarica: The ARM Linker 5l*. 2015. cited page(s) 6, 16, 26, 27, 55, 110

[Pad15c] Yoann Padioleau. *Principia Softwarica: The Text Editor Efuns*. 2015. cited page(s) 31

[Pad16a] Yoann Padioleau. *Principia Softwarica: The Plan 9 C Compiler 5c*. 2016. cited page(s) 52, 111

[Pad16b] Yoann Padioleau. *Principia Softwarica: The Plan 9 Debuggers and Tracers*. 2016. cited page(s) 14, 26, 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