Introduction

The simplest and most obvious approach to writing an emulator is that of a pure interpreter which consists of 3 phases operating upon each instruction in the emulated program:

Read the instruction
Decode the instruction to decide what works need to be done
Perform the appropriate operation

An interpreter that follows the above scheme is inevitably going to be at least tens and probably hundreds of times slower than the machine upon which it is running. The emulated program will however contain a number of loops (sections of code that are executed many times over), and these will be responsible for most of the execution time of the program.

A simple interpreter will therefore read and decode each instruction in the loop many times, yet produce the same result each time - namely "perform this operation upon these operands." The operands will usually be registers

Instruction Cache

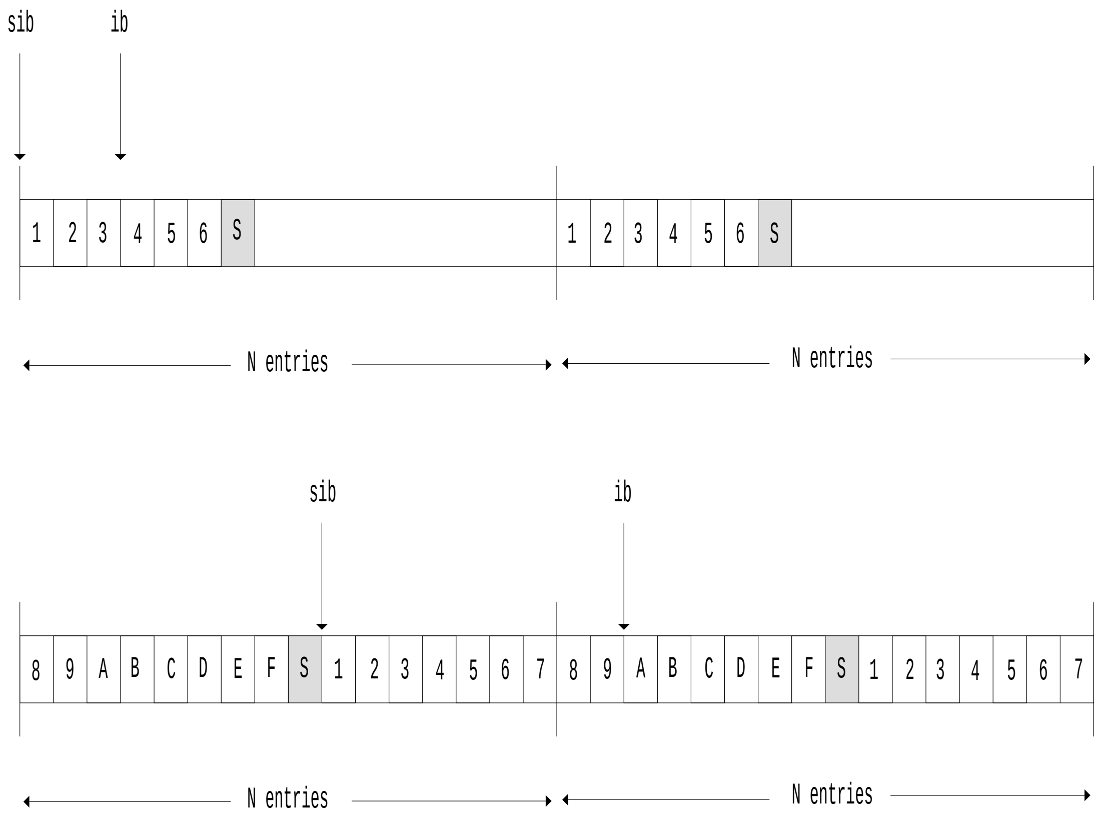
This works in a similar way to the instruction cache of any modern microprocessor, ie. it remembers the instructions that it has accessed recently on the premise that it will need to execute them again soon. The emulator stores the decoded instruction rather than the original ARM instruction and thus manages to avoid the reading and decoding phases if this instruction is executed again.

Cacheing instructions

Instruction Buffer / CodeSequences

The interpreter maintains a number of CodeSequences, each of which is a consecutive run of pre-decoded instructions that have *all* been considered for execution at some point so we "know" that they are valid code. (Note that the instructions may not actually have been executed, oweing to conditional execution)

Each CodeSequence is stored as a double buffer, with each of the n instructions being stored twice at positions x and x + N where N is the 1 greater than the maximum number of instructions that can be stored in a CodeSequence as shown below. The +1 comes about from using a sentinel so that we don't have to check against the end of sequence, we can just keep dispatching and executing sequential instructions until an instruction transfers control to an address other than the next instruction, or the handler code for the sentinel is executed.



The double buffer allows a single pointer to reference the current instruction and to advance linearly without any need to check against another pointer for wraparound.