Bachelor Thesis

Performance Analysis of LLVM and Cranelift Codegen for Just-in-Time Compilation in Emulation

Kevin Axel Kulot
kevin.kulot@ovgu.de

May 17, 2024

First Reviewer:
Prof. Dr. Michael Kuhn

Second Reviewer:
M.Sc. Michael Blesel

Abstract

Just-in-time compilation represents a crucial implementation technique in modern compiler frameworks, as it allows for a continuous analysis of code and architecture-specific optimizations. By combining the flexibility of interpretation with the performance of compiled code, major speedups may be achieved. Emulation, in particular, profits immensely from just-in-time approaches when recreating the behaviour of modern CPUs as naive interpretation will no longer yield satisfactory results.

The balance between generating code quickly and generating performant code, however, remains delicate and can make or break a smooth, stutter-free emulation experience. This thesis presents a hands-on evaluation, comparing the execution and compile time of the code generation process for two state-of-the-art compiler frameworks, each providing a just-in-time implementation for the ARM7TDMI processor of an emulated Game Boy Advance. With each framework suited towards either improved execution time or improved compilation time, that being LLVM and Cranelift respectively, an assessment can be made whether aggressive optimization techniques have a noticeable impact on real-time software, such as emulators. In order to properly utilize the on-the-fly characteristics of dynamic recompilation, caching is used in both implementations to store already-seen compiled code sequences for later use.

Code quality, API usability and general performance were used as evaluation metrics. Even with a limited CPU implementation, the results favor lower compilation time overall despite worse code generation. Cranelift also delivers a more adequate intermediate instruction set, especially since its input represents a real instruction set architecture.

Contents

Chapter 1. Introduction

1.1. Motivation

The preservation of digital media, such as video game consoles and their games, older PC-based hardware or arcade machines, relies on the ability to interact with the medium. Besides archiving the software itself, emulation may be used to replicate the necessary hardware environments. As the complexity of CPUs continues to rise however, with ever-increasing clock speeds and intricate pipelining, emulating at satisfactory speeds has become progressively more difficult. Since most processors are not emulated in isolation but are part of a larger emulation core, additional overhead may arise with the inclusion of other components. These subsystems, such as Picture Processing Units (PPU), Graphics Processing Units (GPU), Audio Processing Units (APU), have to be updated and kept track of alongside the main core, further increasing the computing power needed to keep up with the system's demands.

Historically, emulation was approached by manually interpreting the CPU instructions one-by-one — recreating their behaviour with a high level language. This approach has worked reasonably well for most hardware up to the 16-bit console era but has quickly become a bottleneck for performance. Modern consoles have grown exponentially in hardware complexity; often with multiple CPUs and other coprocessors running in tandem, responsible for things such as vector math instructions, compression algorithms or generating high resolution images. It is no longer feasible to emulate such hardware at full speed without resorting to low-level optimizations — making use of the processor inside the user's computer (host) by generating native instructions on the fly.

A so-called Just-in-Time (JIT) compiler, or dynamic recompiler, is frequently utilized to translate between device (guest) and host instructions during the program's runtime. The emission of host code in the form of basic instruction blocks and the subsequent execution of these blocks as they are compiled on-demand serves as an integral part in this process. This approach leverages the speed of executing assembly code directly at the potential cost of accuracy. Further complexity can be introduced by first translating the guest code into an intermediate representation (IR), performing additional optimization passes, before continuing the translation into host code. This creates an abstraction layer which can be used as a building block for platform independence.

Developing a sophisticated JIT compiler from the ground up is a complex task covering many low-level-development characteristics: understanding the encoding of different instruction sets, familiarity with calling conventions, implementing optimization passes, keeping track of register usage, etc. Compared to a simple instruction interpreter, dynamic recompilation as described above requires careful planning and design. Fortunately, the task of building such a compiler backend is facilitated by resorting to already existing compiler frameworks, namely the LLVM Project[1] or the Cranelift[2] compiler. These frameworks provide an already fully functioning backend with their own IRs, optimization passes and — most importantly — support for most CPU architectures out of the box. This ensures that the focus stays on improving the emulation performance by utilizing their respective APIs while the frameworks do the heavy lifting regarding code generation. As the priority is on running the emulation as fast as possible, the main objective is to balance the speed of generating the code with the speed of the generated code. Cranelift and LLVM focus on each of these respectively.

The goal for this thesis is the implementation and following analysis of two JIT compilers in an emulation context with the help of the aforementioned compiler backends. Both the quality of the generated code but also the resulting performance differences, stemming from two unique state-of-the-art algorithmic approaches in compiler design, are going to be evaluated. The target is the Game Boy Advance (GBA), hosting a 32-bit ARM7TDMI CPU and the ARMv4T instruction set @ 16.78 MHz.

1.2. Structure

This thesis begins by discussing the fundamentals necessary to understand emulation, the architecture of the Game Boy Advance, and why this work is related to it in Chapter 2. Continuing with Chapter 3, the techniques used in both LLVM and Cranelift and how they compare with respect to implementation and their inherent usefulness for emulating a handheld console will be introduced. These consist of storing and caching instructions, allocating the apt registers to maximize throughput, the difference in instruction design and how they handle control flow between blocks, etc. Chapter 4 then will take a look at related work, both historically and contemporary, while also exploring different approaches. Results will be evaluated in Chapter 5: usability, code generation and speed differences will be compared for both compiler frameworks. Finally, Chapter 6 summarizes what this work has achieved and talks about future additions that may be interesting to look at but were out of scope for this thesis.

[1]`https://llvm.org/`
[2]`https://cranelift.dev/`

Chapter 2.

Background

This chapter will begin by exploring the emulation target, the Game Boy Advance, with a short introduction to its instruction set architecture before taking a deeper look into the rest of the system. Following that, the concept of emulation itself, including its applications and common implementation techniques, are going to be defined. The aforementioned emulation techniques will then be explained in more detail with respect to their advantages and disadvantages — along with their relation to the GBA. Finally, two sizable compiler frameworks, LLVM and Cranelift, capable of driving the emulation core by utilizing just-in-time code generation, will be introduced with a focus on their goals and usecases regarding recent compiler advances.

2.1. Game Boy Advance

The Game Boy Advance (GBA), released in 2001, marked Nintendo's first foray into 32-bit handheld consoles. Compared to its 8-bit predecessor, the GBA was based on a modern RISC (Reduced Instruction Set Computing) CPU but included the processor of its previous iteration for backwards compatibility. Furthermore, it was announced "that the GBA would be a dedicated high-quality two-dimensional game platform, bypassing any mistaken hopes for a 3-D system" solidifying the focus on 2D graphics. This is accompanied by an overall increase in on-board and video memory [Granett, 2000].

With a 2.9 inch reflective TFT color LCD screen running at a resolution of 240×160 pixels and a maximum of 32,768 colors, a graphics engine similar to the Super Nintendo Entertainment System (SNES), featuring multiple background layers, a sprite layer and special color effects such as alpha blending or affine transformations [Copetti, 2019, see *Graphics*], balancing both cycle-accurate and fast emulation had become more difficult. Section 2.1.1 intents to provide a deeper understanding of the system instruction set architecture and its corresponding processor before Section 2.1.2 analyzes the rest of the system internals. The following section is mostly based on the data sheet [Advanced Risc Machines Ltd., 1995] and technical reference manual [ARM Limited, 2001].

2.1.1. Introduction to ARMv4T Assembly

The CPU inside of the GBA is the ARM7TDMI: a 32-bit general purpose RISC microprocessor, meaning the instruction set decoding mechanisms are simpler resulting in lower power consumption while keeping performance high [ARM Limited, 2001, p. 22]. To be more precise with naming, the processor designation is concatenated with additional letters signifying the presence of certain features. Table 2.1 shows the features each of the respective letters enable. With regards to the work presented in this thesis, only THUMB Mode and Long Multiply will be needed for a working CPU implementation leading to two fixed-width instruction sets — 32-bit ARM and 16-bit THUMB — and an extra ARM instruction.

`T`	**T**HUMB MODE	Second super reduced 16-bit instruction set (v1)
`D`	JTAG **D**EBUG	Industry standard for verifying designs and testing printed circuit boards after manufacture
`M`	LONG **M**ULTIPLY	Inclusion of long multiplication of two 32-bit operands to produce a 64-bit result (`MULL`, `MLAL`)
`I`	EMBEDDED**I**CE	Internal on-chip unit to force debug state / halt

Table 2.1.: Naming of ARM processor variants [Analog Devices Inc., 2005].

A three stage pipeline strategy is employed on the ARM7 allowing the processor to work on different parts of multiple instructions simultaneously. These three stages consist of the following steps (in order): Fetch → Decode → Execute. As one instruction is being executed, its successor is being decoded and a third one fetched from memory [ARM Limited, 2001, p. 23]. Each stage in the pipeline is overlayed with a different one working on the next instruction as Figure 2.1 demonstrates. This also reflects on the usage of certain registers during execution time and needs to be considered when developing for this architecture. Specifically, the *program counter* (PC), or `R15`, always points to the instruction currently being fetched leading to its value being two instructions ahead of the currently executing one when read. Coincidentally, this workflow overlaps reasonably well with the common structure of an interpreter loop used in emulation, see Section 2.2.2.

The ARM7TDMI has 31 general purpose 32-bit registers and six status registers. However, not all of them are available at the same time as the current register set depends on both the operating mode and operating state. The seven operating modes are as follows [Advanced Risc Machines Ltd., 1995, p. 30]:

- **User:** Normal ARM program execution state.
- **FIQ:** Supports data tansfer (Fast Interrupt Request).
- **IRQ:** General-Purpose Interrupt Handling.
- **Supervisor:** Protected mode for the operating system (BIOS).
- **Abort mode:** Data / Instruction Abort.
- **Undefined:** Entered when an undefined instruction is executed.
- **System:** Privileged User Mode.

Each operating mode has its own register bank with up to 16 general purpose registers visible at a time in ARM state and 11 in THUMB state[3]. Furthermore, operating modes have their own banked stack pointer (`R13`), link register (`R14`) and Saved Program Status Registers

[3]`R0` - `R7` are general purpose; `SP`, `LR`, `PC` have a special purpose each.

(SPSR) — except FIQ mode which uses `R8` - `R14` as its banked register set. Figure 2.2 showcases the banking behaviour. Similarly, THUMB state shares this behaviour but only for the stack pointer and link register as it only has direct access to eight general purpose registers (`R0` - `R7`) without the use of special instructions.

Operating modes, states, information about flags from arithmetic or logical instructions, whether interrupts are enabled, etc., are stored in a special register, named *Current Program Status Register* (CPSR), shared across all modes. With the exception of one instruction family, this register cannot be modified directly by the programmer, rather it is usually affected by

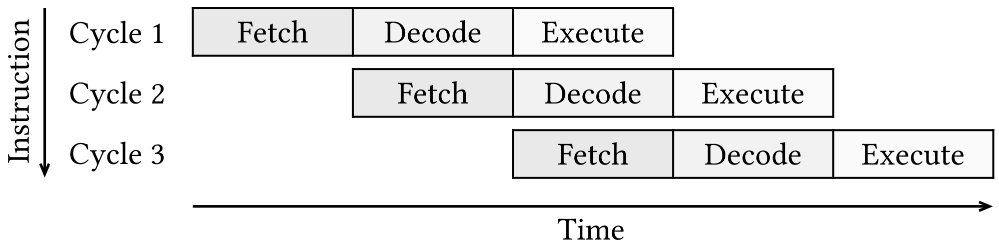


Figure 2.1.: Three-Stage Pipeline.

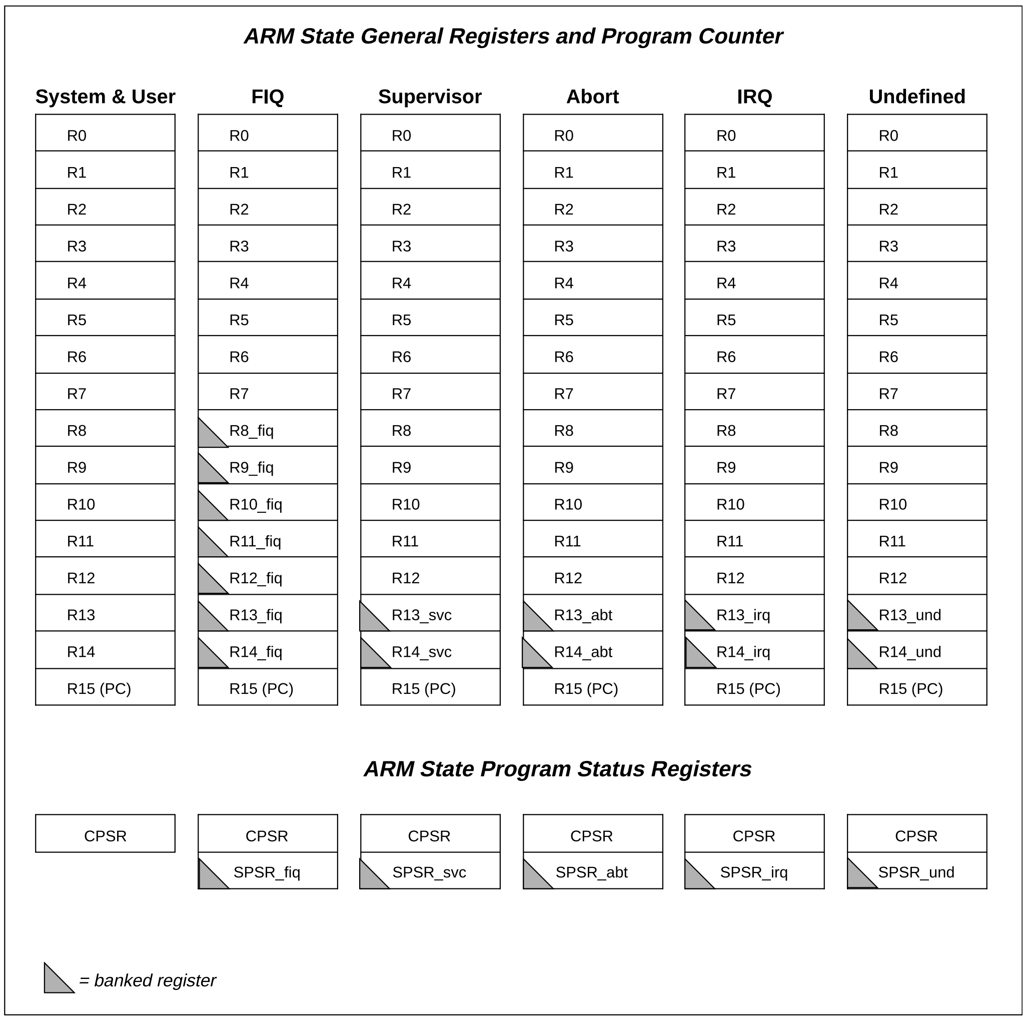


Figure 2.2.: Register organization in ARM state [Advanced Risc Machines Ltd., 1995, p. 31]. Banked registers are private to the operating mode while the others are shared.

ALU operations leading to overflows, zero results, signage or carry changes. Additionally, switching state, which is usually done with the `BX` instruction, or any kind of mode change either through exception handling or manual assembly will affect CPSR (Figure 2.3).

All ARM instructions may be executed conditionally. Bits 31–28, as seen in Figure 2.4c, specify the condition based on the current condition code flags of CPSR. In theory, this leads to 2^4 different combinations of conditions, however, in practise one combination (`0b1111`) is reserved and shall not be used. Each condition is represented by a suffix which can be appended to the instruction mnemonic, for example, `MOV` becomes `MOVEQ` indicating data will only be moved if the `Z` flag is set. Bit 20, the `S`-Bit, is used in every ALU instruction and indicates whether or not the result of the operation sets the corresponding condition codes in CPSR. Other important bits include the `I`-Bit for deciding if an operand is an immediate value or a register, the `L`-Bit to determine if a data transfer stores to memory or loads from memory and `P`, `U`, `W` all leading to various data transfer variations. Unlike register-memory architectures like x86 where operands of an operation may be in memory or in a register, ARM is a load-store architecture dividing the instruction set into two categories: memory access and ALU operations. Both operands and the destination location must reside in registers. Most instructions use bits 19–16 for the source or base register (`Rn`) and bits 15–12 for the destination register (`Rd`).

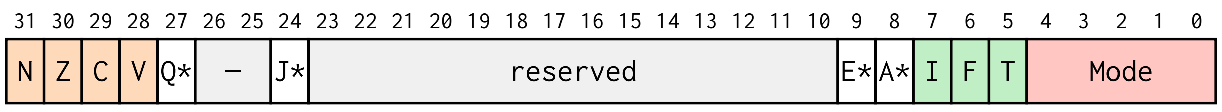
Figure 2.4 shows an example of this encoding: the *Data Processing* instruction family features the condition field, the `I`-Bit to determine the value of `Operand 2`, the aforementioned `S`-Bit and both a source and destination register. The precise ALU operation is specified within the `Opcode` field (Bit 24–21). Figure 2.4a and Figure 2.4b further demonstrate another feature of the ARM7TDMI with respect to the `I`-Bit: the barrel shifter unit. It is responsible for modifying `Operand 2` either via shifting the value in the given register or via rotating the given 8-bit immediate value. Immediate operand rotates enable the generation of common constants, such as powers of 2, through usage of the `Rotate` field — `Imm` gets zero extended to 32 bits and is then rotated right by $2 \cdot$ `Rotate` — as seen in Listing 2.1.

```rust
// Extract `Rotate` field and truncate to 4 bits.
let ror = (op >> 8) & 0xF;
// Extract `Imm` field and apply rotate right.
let res = (op & 0xFF).rotate_right(2 * ror);
```

Listing 2.1.: Implementation of immediate operand rotation with Rust. Encoding is given via the variable `op` as an unsigned 32-bit integer.

Register shifts are controlled by the `Shift` field which denotes the shift type and shift amount. Possible shift types are: logical shift left (`LSL`), logical shift right (`LSR`), arithmetic shift right (`ASR`), rotate right (`ROR`). The shift amount may be specified either as a 5-bit unsigned immediate or inside of a shift register (`Rs`). All barrel shifter operations affect the Carry flag of CPSR, usually by setting it to the discarded bit value. Exceptions to this rule are special encodings, for example, the shift amount being 0 or (more than) 32 leading to special behaviours regarding both the result and carry out, see [Advanced Risc Machines Ltd., 1995, pp. 54–57].

All previously mentioned features result in a very dense ISA and a rather complex decoding step but ensure that the design stays within the RISC philosophy of less work per instruction.



*not supported on the ARM7TDMI.

🟧 Cond. Code Flags (**N**: Sign, **Z**: Zero, **C**: Carry, **V**: Overflow)

🟩 Misc Control Bits (**I**: IRQ Disable, **F**: FIQ Disable, **T**: State Bit)

🟥 Mode Control Bits (Bit 4 is always set, order as above)

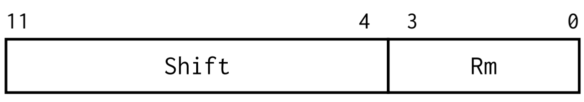
Figure 2.3.: Representation of the Current Program Status Register.

Moreover, it is possible to *interrupt* the normal control flow of executing code by means of external hardware signals, so-called hardware interrupts, or manually through so-called software interrupts with the `SWI` instruction. Hardware interrupts cause the processor to enter IRQ mode and jump to the corresponding IRQ vector, that is, set PC to `0x18`. Which external signals actually cause an interrupt request (henceforth abbreviated as IRQ) is decided by the rest of the Game Boy Advance hardware and controlled through two I/O registers. The Interrupt Enable (`IE`) register and Interrupt Request Flags (`IF`) register have to match in order to dispatch an interrupt and enter its routine. Furthermore, bit `0` of the Interrupt Master Enable (`IME`) register has to be set and the `I`-Bit in CPSR has to be cleared.

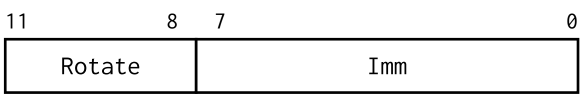
A software interrupt, on the other hand, may simply be dispatched when `SWI` is encountered. The CPU, then, enters SVC mode and jumps to address `0x08` which lies inside of the BIOS memory space. Therefore, software interrupts can be thought of as system calls, often used by GBA software for common routines such as LZ77 compression, Huffman encoding, Run-length encoding or more involved arithmetic like division, square roots and $\arctan$.

When an interrupt is serviced, or the program flow is halted through other means, the processor enters an exception. Exceptions are handled by preserving the address of the next instruction in the appropriate link register, copying CPSR into the SPSR of the mode the CPU is entering, changing the CPSR mode bits and jumping to the corresponding exception vector. Leaving an exception should be done through moving the link register back into the program counter, copying SPSR back into CPSR and optionally clearing the IRQ disable flag.

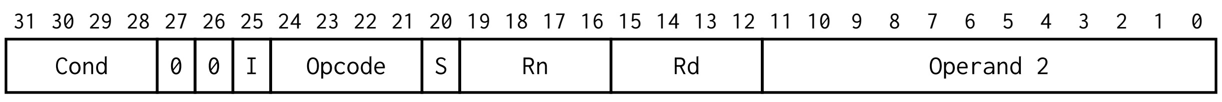
Section 2.1.2 will go into more detail with regards to hardware interrupt sources, how GBA software utilizes the BIOS and processor ↔ hardware communication. Memory mapping, LCD timings and other hardware idiosyncrasies will also play an important role in this chapter.



(a) Meaning of Operand 2 when `I = 0`.



(b) Meaning of Operand 2 when `I = 1`.



(c) Full 32-bit encoding.

Figure 2.4.: Encoding of the *Data Processing* instruction showing the information density per opcode, recreated from [Advanced Risc Machines Ltd., 1995, p. 52].

As previously mentioned, the ARM7TDMI also supports a second 16-bit instruction set, called THUMB, enterable via the `BX` instruction when Bit 0 of the operand register is set. By implementing a 16-bit instruction length on a 32-bit architecture, processing of 32-bit data is made more efficient and compact with better code density. As THUMB code operates on the same register set, most of ARM's performance is retained. According to ARM, "THUMB code is able to provide up to 65% of the code size of ARM, and 160% of the performance of an equivalent ARM processor connected to a 16-bit memory system" [Advanced Risc Machines Ltd., 1995, p. 10]. Besides conditional branches, THUMB instructions are always executed regardless of the condition codes.

Ultimately, both the ARM and THUMB instruction set (see Appendix A.1 and Appendix A.2) effectively end up having the same effect on the processor as the Instruction Decoder Unit internally maps each THUMB instruction to the equivalent ARM instruction, with the exception of *long branch with link*, before entering the execution step. Usually, the ARM7TDMI may also include a coprocessor (CP14) extending its functionality, though in the case of the GBA, CP14 is not connected and therefore its instructions can be ignored.

2.1.2. System Architecture Overview

In addition to the main processing unit, the rest of the system's architecture also fulfills an important role to ensure the handheld stays "simple yet complete" while also keeping power consumption low [Granett, 2000]. Figure 2.5 depicts how the rest of the system is structured, how the components are connected together via a memory bus and how they communicate with the CPU. Most sections in this chapter will be based on this figure.

The Game Boy Advance uses memory mapped I/O (MMIO) which means that the same address space is used to refer to both main memory and I/O devices. There are no special I/O instructions needed to write or read from external devices or ports — if data is to be fetched or stored at memory address X, it automatically maps it to the corresponding I/O device if X lies inside the specified memory range of the device [Hayes, 1978]. Therefore, inter-component communication may be realized through reads and writes from the CPU inside of certain memory ranges. The process of assigning these memory ranges, or memory regions, to specific devices or I/O registers is called *memory mapping* and leads to a so-called memory map. Each device inside of the GBA and their assigned memory regions have a specific purpose — though not all regions are fully readable or writable (Appendix A.3).

CPU & Bus Behaviour. Unable to read from misaligned addresses, the ARM7TDMI forcibly aligns the used address if necessary. In case of 32-bit reads/writes, the address has to be aligned at a boundary of 4 bytes — meaning it has to be divisible by 4. For 16-bit reads/writes, the same requirement applies but at a boundary of 2 bytes. It does so by using the rounded-down memory address, ignoring the low bits[4] and optionally rotating the read data for certain instructions[5]. Branches and other instructions that may modify the program counter also have to ensure correct alignment of PC as to avoid spurious program execution which can cause the CPU to interpret game data as instructions or generally coming across instruction encodings

[4] `ARM`: Ignore Bit 0 and Bit 1, `THUMB`: Ignore Bit 0.

[5] `LDR` and `SWP` rotate the data as follows: `rotr data, (addr & 3) * 8`.

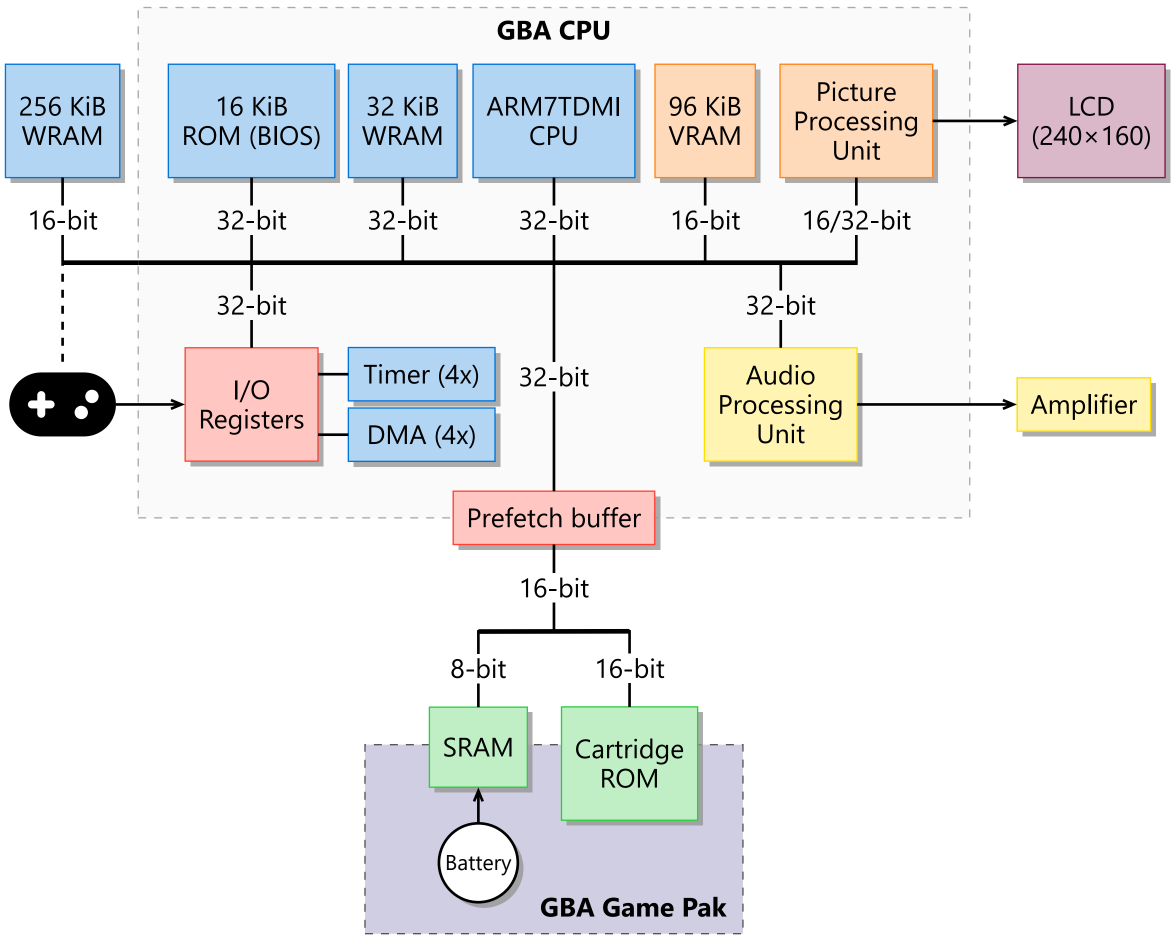


Figure 2.5.: Abridged overview of the GBA architecture, based on [Copetti, 2019].

it cannot handle. This would then generate an *Undefined Instruction Exception*, triggering the internal exception routine which involves jumping to the corresponding exception vector and entering the **Undefined** operating mode on entry.

Additionally, many addresses point to unused memory either within the used address ranges or outside of them. Special rules apply depending on the exact location:

- Most internal memory is mirrored across the address space in which it is located.
  - `WRAM` is accessable throughout `0x02XXXXXX` / `0x03XXXXXX`, repeats at 256/32 KiB.
  - `PRAM` is accessable throughout `0x05XXXXXX` and repeats each `0x400` bytes.
  - `VRAM` is accessable throughout `0x06XXXXXX` and repeats each `0x20000` bytes.
  - `OAM` is accessable throughout `0x07XXXXXX` and repeats each `0x400` bytes.
- Reading from non-mirrored unused memory leads to *open bus* behaviour.
  - Invalid reads return the last value given to the CPU from that bus.
  - Due to pipelining, the instruction being fetched from memory is a few bytes ahead of where the currently executing one is located (fetched at the beginning).
  - This value is then returned by the *open bus*.

Some subtle differences have been discovered with regards to where the instruction was executed from affecting the precise open bus value[6] though the general idea remains the same.

[6] `https://www.ngemu.com/threads/gba-open-bus.170809/` (accessed February 22, 2024)

BIOS & Startup Routine. The first 16 KiB of memory inside the GBA are reserved for the BIOS, or system boot ROM, and may only be read from if the program counter is also located within the BIOS area. Writes are prohibited and do nothing. It is responsible for initializing the stack pointers and the Saved Program Status Registers of all operating modes, verifying the legitimacy of the inserted cartridge by performing a byte for byte check of the Nintendo® logo as its presence is required inside the cartridge header, displaying the “GAME BOY” intro sequence and providing the `SWI` functions first mentioned in Section 2.1.1.

If no cartridge is present, the startup routine is still performed but the Nintendo® logo will not be displayed, indefinitely leaving the “GAME BOY” text on screen until the device is turned off. Furthermore, the BIOS includes a multiboot functionality allowing the GBA to boot from other consoles or computers — without an inserted cartridge — through its link port and a custom transmission protocol.

System calls are invoked through software interrupts generated from the `SWI` instruction. In ARM state, this instruction features a 24-bit comment field which is ignored by the processor as only bits 27–24 are relevant for proper decoding and identification. Instead, it may be used as an index into a lookup table of entry points for the corresponding BIOS routines. Of those 24 bits, only the upper 8 bits are used as an index, whereas in THUMB state the lower half of the 16-bit opcode is used to encode the requested routine, see Appendix A.2. Therefore, invoking `SWI` number `0x06`, for example, which represents the `Div` function, may look as follows:

```
@ ARM state.   Uses upper 8 bits of 24 bit comment field.
swi #0x060000  ; → swi NN * 0x10000
@ THUMB state. Uses lower half of opcode encoding.
swi #0x06      ; → swi NN
```

Listing 2.2.: Specification of the system call index in ARM and THUMB state where `NN` denotes the index in hexadecimal notation.

Generally, registers `R0` - `R3` are used as input parameters with `R0`, `R1` and `R3` also acting as output parameters, containing either return values or unpredictable data, i.e. the output of the calculation or intermediates. Continuing with `Div` (Signed Division) as an example:

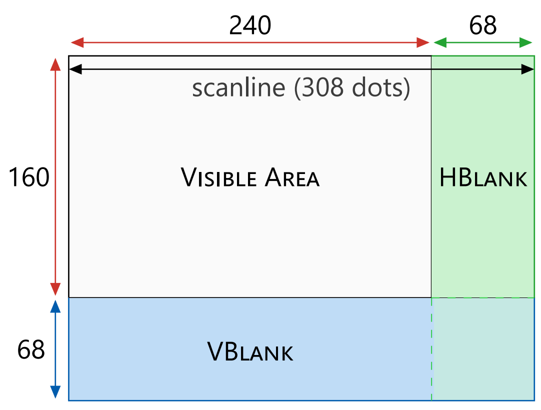
- **Input:**
  - R_0 as signed 32-bit integer numerator, e.g. $R_0 = -1234$.
  - R_1 as signed 32-bit integer denominator, e.g. $R_1 = 10$.
- **Output:**
  - $R_0 = R_0/R_1 = -1234/10 = -123$
  - $R_1 = R_0 \bmod R_1 = -1234 \bmod 10 = 4$
  - $R_3 = |R_0/R_1| = |-1234/10| = 123$

As illustrated, the `Div` routine uses a range of pre-defined registers for both incoming and outgoing parameters; most other arithmetic routines only return one result while non-arithmetic ones do not have a return value. The remaining BIOS functions can be categorized into rotation / scaling functions for calculating the parameters used in the affine transform for backgrounds and sprites, decompression functions such as Huffman decoding or LZ77 decompression, memory copies, halt functions to pause the CPU until a certain event occurs, reset functions and sound driver functions.

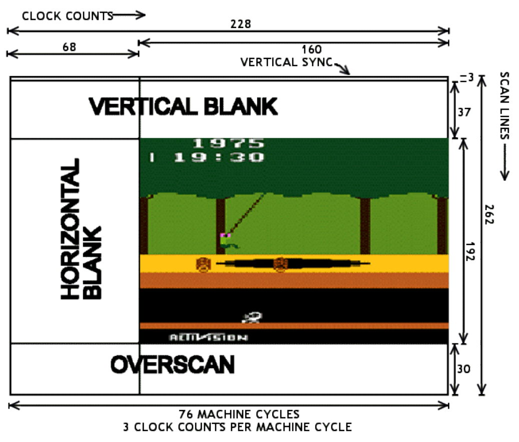
Picture Processing Unit. In order to draw pixels on screen, the Picture Processing Unit (PPU) utilizes a plethora of traditional systems and contemporary rendering techniques. Similar to cathode ray tube (CRT) displays and other analog video technologies, the GBA's LCD screen uses a form of raster scanning to subdivide its output into a sequence of scanlines. In raster scanning, a "beam" sweeps left to right across the display producing a line or row of image elements before entering a blanking stage and returning back to the left to start the next line below [Jack and Tsatsoulin, 2002, p. 242]. CRTs used electron beams which were physically "moved across the screen either by deflection plates or magnets" [Jack and Tsatsoulin, 2002, p. 44] hence the blanking stage or period in which upon reaching the right-hand limit of a scanline, or the very end of the frame, the video signal is blanked as to not reveal the return path of the beam [Jack and Tsatsoulin, 2002, p. 31].

Each scanline has a horizontal blanking period, each frame a vertical blanking period. During those periods, software may modify how the display is generated which can be used to create otherwise unobtainable graphics effects. Figure 2.6 shows that, even though the LCD of the Game Boy Advance no longer used CRT technology, blanking periods were still implemented and lines were drawn from left to right to make use of raster scanning. Ergo, games on this handheld were able to perform calculations outside of the active rendering phase — between scanlines — resulting in such effects as faux or pseudo 3D by applying an affine transformation with different parameters per scanline onto the background layers.

First seen on the Super Nintendo Entertainment System (SNES) under the moniker "Mode 7" [Next Generation, 1996], several games made use of per-scanline transforms to create different depth effects. Mario Kart™ Super Circuit™, for example, relies on this perspective trick for its race tracks, similar to the horizontal number plane in Figure 2.7b. Accomplishing these results requires the software to know *when* the blanking periods happen. This is achieved through hardware interrupts, first introduced in the previous chapter, which the PPU requests whenever a blanking period is entered. After 240 dots (here: pixels, 960 cycles), when the



(a) GBA blanking periods.



(b) CRT blanking periods. [Wright, 1979]

Figure 2.6.: Comparison of blanking period lengths between a GBA and an ATARI 2600.

visible area of a scanline has finished rendering, the horizontal blanking period begins and lasts another 68 dots. This causes the hardware to toggle the H-Blank flag in the Display Status Register, and, if the H-Blank IRQ Enable flag is set within the same register, to request and possibly service an interrupt. Likewise, when the visible area of a frame has finished rendering after 160 scanlines, the vertical blanking period begins and lasts for another 68 lines. The V-Blank flag is toggled and the V-Blank IRQ Enable flag controls whether an interrupt is requested.

The PPU also handles a third interrupt, unrelated to blanking, called V-Counter IRQ. To indicate which scanline is currently being drawn, a Vertical Counter register (VCOUNT) is incremented each line, including hidden ones during V-Blank. Furthermore, the Display Status Register features a V-Count setting with an identical range to VCOUNT of $[0, 227]$. Should the values of VCOUNT and V-Count setting match, the V-Counter flag is toggled and, if enabled, its interrupt requested. All three flags, H-Blank, V-Blank, V-Counter, and VCOUNT are read-only and may only be modified through hardware.

A frame consists of multiple layers. Each individual layer can be enabled or disabled and arranged through priority settings. Frames can be composed with up to four background layers and one sprite layer, also called objects or OBJ, and then further modified with additional graphics effects, as seen in Figure 2.7. Background layers may either be regular or affine depending on the selected video mode. Regular backgrounds, also known as Text Mode, place 8×8 tiles onto the frame, taken from a tile map which specifies a mapping from a tile ID to the matching pixel data in VRAM. A tile map can be visualized as a screen-sized grid of tile entries, 16 bits in size, responsible for storing a tile ID, whether it is horizontally or vertically mirrored, and an optional palette index. Calculating the memory address of the pixel data also requires two additional constants per background: Screen Base Block (SBB) and Character Base Block (CBB). The SBB specifies an offset within the tile map data, the CBB specifies an offset within the tile pixel data. Both offsets are configured through the BG**x**CNT registers (Background **x** Control) where $\mathbf{x} \in [0, 3]$; each background has its own control register.

Pixels may either have a color depth of 4-bit or 8-bit[7]. If a pixel is encoded as having a color depth of 4-bit, its color is part of a 16 color palette inside of palette RAM. Palette RAM may feature up to 16 different palettes with 16 colors each. The aforementioned palette index selects the palette while the pixel data specifies the color. 8-bit depth allows up to 256 colors but does not make use of palettes, rendering the palette index useless. Colors on the GBA are encoded with the RGB555 format, thus each color component consists of 5 bits with no alpha channel, resulting in a total of 2^{15}, or 32768, possible colors. Nonetheless, palette RAM stores each color as a 16-bit value.

Affine backgrounds, on the other hand, utilize extra rotation/scaling parameters in order to first transform screen space into texture space, via a transformation matrix, before applying the same tile to pixel data mapping. Vertical or horizontal mirroring of tiles is not possible as the tile entry only consists of a 7-bit tile number; additionally affine backgrounds always use a color depth of 8-bit. While the hardware itself does not use multiplication to apply this effect, it may be helpful to think of it as a matrix-vector multiplication where the matrix is

[7]In tilemap-based modes and Mode 4. Mode 3 and 5 encode pixel data differently.

made up of the rotation/scaling parameters and the vector represents the screen coordinates, as in Equation 1, which is semantically equivalent.

$$
\begin{aligned}
&1.\ \text{Let } \mathbf{A} = \begin{pmatrix} \mathrm{pa} & \mathrm{pb} \\ \mathrm{pc} & \mathrm{pd} \end{pmatrix} \text{ be the affine parameters.} \\
&2.\ \text{Let } \mathbf{x} = \begin{pmatrix} x \\ y \end{pmatrix} \text{ be the screen space coordinates.} \\
&3.\ \text{Let } \mathbf{y}_{\mathrm{tex}} = \mathbf{A}\mathbf{x} = \begin{pmatrix} \mathrm{pa} & \mathrm{pb} \\ \mathrm{pc} & \mathrm{pd} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \mathrm{pa} \cdot x + \mathrm{pb} \cdot y \\ \mathrm{pc} \cdot x + \mathrm{pd} \cdot y \end{pmatrix}. \\
&\quad \text{Thus, } \mathbf{y}_{\mathrm{tex}} \text{ represents a transformation mapping from } \mathbb{R}^2 \text{ to } \mathbb{R}^2, \\
&\text{and } \mathbf{A} \text{ represents the corresponding transformation matrix.}
\end{aligned}
\tag{1}
$$

Calculating the memory offset into VRAM needed to retrieve the tile ID would then be carried out with $\mathbf{y}_{\mathrm{tex}}$ instead of the regular screen coordinates. Possible geometric transformations include: Rotation, Shearing, Stretching, Squeezing and Reflection. Figure 2.7e and Figure 2.7f demonstrate rotation and stretching respectively.

Table 2.2 provides an overview of how the selected video mode determines the frame composition and background attributes. Mode 0, for example, allows up to four regular backgrounds, 4-bit and 8-bit color depth and all special graphics effects. Mode 1 only uses three background layers of which the first two are regular and the last one allows for affine transforms. Mode 2 exclusively uses affine background layers, though it is limited to layer 2 and layer 3 only with no support for 4-bit color depth. Modes 3 - 5 behave differently as they are not based on tiles and tile mapping but act like a direct frame buffer rather. As such, every two bytes in VRAM directly correlate to a 15-bit color value — with 480 bytes defining exactly one scanline — except for Mode 4 which still utilizes palette RAM. Mode 4 associates one byte with each pixel, selecting one of the 256 possible palette entries and thus using 240 bytes per line.

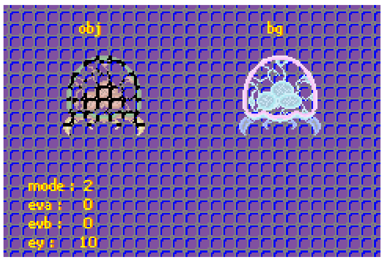
Finally, sprites are always drawn regardless of video mode as long as they are enabled in the Display Control Register. A sprite object is made up of OBJ attributes, each 16 bit in size, three per object, located in Object Attribute Memory (OAM). These attributes determine the

MODE	ROT/SCAL	LAYERS	SIZE	TILES	COLORS	FEATURES*
0	No	`0123`	256×256..512×512	1024	16/16..256/1	`SFMABP`
1	Mixed	`012-`	*BG 0 & BG 1 as Mode 0, BG 2 as Mode 2.*			
2	Yes	`--23`	128×128..1024×1024	256	256/1	`S-MABP`
3	Yes	`--2-`	240×160	1	32768	`--MABP`
4	Yes	`--2-`	240×160	2	256/1	`--MABP`
5	Yes	`--2-`	160×128	2	32768	`--MABP`

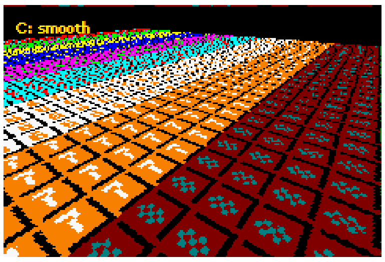
*FEATURES: **S**crolling, **F**lip, **M**osaic, **A**lpha Blending, **B**rightness, **P**riority.

Table 2.2.: Summary of the facilities of the separate background modes. Modes 0 - 2 are tilemap-based, modes 3 - 5 are bitmap-based, mode 4 and 5 can utilize double buffering [Korth, 2001, see *LCD I/O Display Control*].

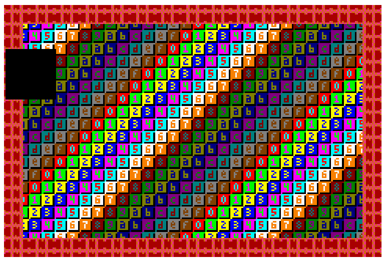
sprites x-coordinate, y-coordinate, color depth, priority, tile number, which set of rotation/ scaling parameters to use, etc (Appendix A.4). As OAM has a size of 1024 bytes and each entry is 48 bits in size, this leaves 128 16-bit gaps which are filled with 32 sets of four 16-bits rotation/scaling parameters each, amounting to a maximum of 128 sprites. Regular and affine sprites operate in a similar way to their background counterparts with slightly different flag behaviors (Mirroring vs. Geometric Transform, Disable vs. Double-Size) determined by the `Rot/Scale` flag. In both cases, however, the PPU may utilize the transformation matrices so by default, all are set to the identity matrix as to not affect regular sprites.



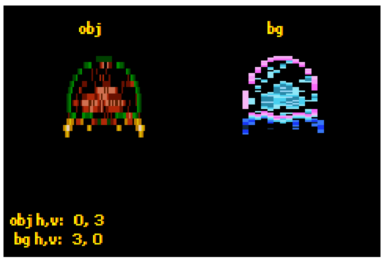
(a) Alpha Blending.



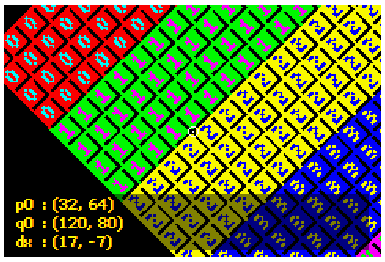
(b) Pseudo-3D.



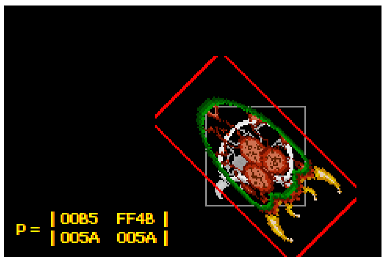
(c) Window Layers.



(d) Mosaic.



(e) Affine Transform (BG).



(f) Affine Transform (OBJ).

Figure 2.7.: An assortment of possible hardware graphics effects [Vijn, 2005].

In conclusion, the Game Boy Advance PPU builds upon its predecessor, expanding its capabilities where necessary but keeping its core structure similar. Special hardware graphics effects such as Alpha Blending, Brightness Adjustments or Mosaic (Figure 2.7a and Figure 2.7d) combined with an overall increase in video memory and screen resolution make the PPU a capable but complex graphics chip.

I/O: Timer & DMA. Timing is done by including four 16-bit incrementing timer units. Each timer may be configured through a Timer Control Register which determines whether the timer is enabled, if it should trigger an interrupt and its increment frequency. Additionally, all timers store a counter and reload value. Possible frequencies include: 16.78 MHz (every cycle), ~0.262 MHz (every 64 cycles), ~65546 kHz (every 256 cycles), ~16386 kHz (every 1024 cycles). A timer may also utilize the Count-Up setting whereas instead of incrementing at a set frequency, it is incremented whenever the preceeding timer overflows. On timer overflow, its counter gets reloaded with the reload value and, if timer IRQs are enabled, requests the corresponding interrupt.

DMA, or Direct Memory Access, describes “a form of data transfer which allows data to move between microcomputer memory and external logic without involving the microprocessor in the data transfer” [Osborne, 1980, p. 5–18/166]. A device with this capability is usually called a DMA Controller of which the GBA has four; sometimes also referred to as DMA Channel. During ongoing DMA transfers, the CPU, as well as DMA channels with a lower priority, are paused. This happens because the DMA Controller takes control of the bus while it is active, effectively making it a “cycle stealing” DMA [Osborne, 1980, p. 5–70/219]. A DMA Channel may be configured through a DMA Control Register, mirroring the design of timers. Furthermore, each channel needs a source address, a destination address and how many data units shall be transferred. The size of data units depends on the transfer type, though it can either be 16-bit or 32-bit per unit. All units are transferred by sequential reads and writes, meaning roughly one unit each cycle, and once a transfer is finished, an IRQ may be requested.

In order to transfer as much data as possible in an efficient manner, a DMA channel also has the ability to increment or decrement both the source and destination address after the transfer of each data unit. Should special effects each scanline or frame be desired, modifying a channel’s start timing may be an option as it can be immediate, at H-Blank or at V-Blank, repeating if necessary. Ultimately, the purpose of a DMA Controller is to transfer data much faster — in the least amount of cycles possible — than a CPU could with manual assembly.

Cartridge / Game Pak. Games and other software must be stored inside of a ROM cartridge, or colloquially game cartridge, and “may also include [...] backup medias, used to store game positions, highscore tables, options, or other data” [Korth, 2001, see *GBA Cartridges*] even when the main device is powered off. This extra cartridge RAM may use either Static RAM (SRAM), Ferroelectric RAM (FRAM), EEPROM or Flash ROM depending on the manufacturer, and can hold anywhere between 32 KiB and 128 KiB of memory.

ROM sizes varied between 4 MiB and 32 MiB. The first 192 bytes inside of ROM memory space make up the cartridge header consisting of the entry point, the Nintendo® logo, the game title, game code, maker code, etc [Korth, 2001, see *GBA Cartridge Header*]. Some cartridges also feature a Real Time Clock (RTC) to make use of day-night cycles in games or other real-time based events. Other add-ons include a solar sensor, a tilt sensor, a gyro sensor, rumble monitors and an e-reader for dotcodes.

2.2. Fundamentals of Emulation

2.2.1. Concept and Application

Emulation as a technique for replicating hardware behaviour has been around since the 1960's when it first became necessary to speedup the process of simulating IBM mainframes with microcode. This concept of using additional microcode or other hardware assistance in addition to a simulator led to such significant performance improvements that it was subsequently given the name “emulator” [Pugh, 1995, p. 274]. Back then, “emulation” mostly referred to hardware-assisted simulation whereas the term “simulation” referred to pure software-based solutions [Tucker, 1965]. Nowadays, the term emulation may refer to both hardware-based emulation and software-based emulation.

In essence, an emulator seeks to imitate a certain computer architecture (guest) within another platform (host). As hardware may fail over time and software developed specifically for this hardware becomes obsolete, accessibility to both is reduced significantly. Similarly, certain older hardware devices, including most of their software catalog, might no longer be available for purchase with no more active manufacturing occuring either. Digital preservation through emulation seeks to mitigate this availability/service issue, as it allows for a proper recreation of the sought after look and feel, or functionality, of the target system [Koninklijke Bibliotheek, 2006].

On a technical level, emulation may be realized either through interpretation (see Section 2.2.2) of the processor instruction set or just-in-time compilation (see Section 2.2.3) of guest instructions with the host processor.

2.2.2. Interpretation

In programming language development, an interpreter represents an implementation technique which takes in a form of source code and executes it immediately, running programs "from source" [Nystrom, 2021]. After parsing, an interpreter may either walk through its Abstract Syntax Tree (AST) directly and evaluate the instructions and expressions found within or translate its source into an intermediate representation, optionally performing optimizations, executing the IR one-by-one. This is commonly referred to as a tree-walk interpreter and bytecode interpreter respectively. Alternatively, a CPU can also be thought of as an interpreter of precompiled machine instructions; similar to a bytecode interpreter.

When referring to interpretation in an emulation context, no parsing or lexing of source code is needed. Instead, machine instructions of the guest platform are turned into semantically equivalent high-level code which then executes immediately. [Tucker, 1965] first talked about this *conversion problem* when he proposed "simulation" — that is, software-based emulation via interpretation — as a potential solution. His approach describes the now common concept of interpretation in emulation: Each instruction is interpreted in sequence at execution time while an "image of the entire state of the machine being simulated is maintained". It is, therefore, only necessary to alter the machine state as the instruction would have done to ensure correct behaviour. However, [Tucker, 1965] also states that an interpreter must interpret an instruction each time it is executed leading to large performance degradations due to interpretive overhead, demonstrating a potential trade-off between accuracy and performance.

Figure 2.8 demonstrates this process applied to the ARM7TDMI processor, incidentally following a similar fetch-decode-execute routine as the three-stage pipeline outlined in Figure 2.1. First, the current CPU state has to be evaluated in order to use the appropriate instruction set. This is done with Bit 5, or the T-Bit, of the Current Program Status Register. If the T-Bit is set, the CPU is in THUMB state and, thus, 16 bits are to be fetched from memory representing a THUMB opcode encoding. If the T-Bit is cleared, the CPU is in ARM state and a 32-bit ARM opcode is fetched from memory. Decoding, described in more detail in the following paragraphs, follows a similar strategy regardless of instruction set based on either pattern matching through multiple nested `switch`-statements or lookup tables. The execution step is then responsible for adequately translating the semantics of the opcode into high-level code;

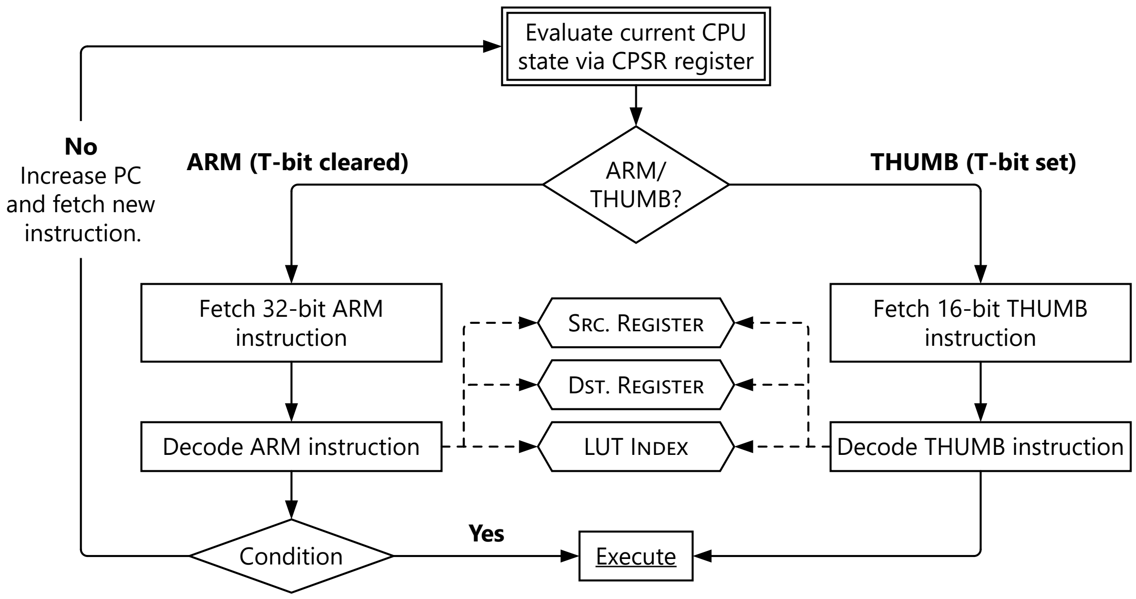


Figure 2.8.: Interpreter Emulation Loop. The hexagonal nodes represent values extracted from the decoding step. *Condition* represents both a decoded value as well as the actual check for condition codes in CPSR.

at least to the extent that most software should not be able to tell if it is running on real hardware or an emulator. The following paragraphs intend to define each step in more detail with regards to an ARM-based system specifically, based on the implementation developed for this thesis in particular.

Fetch. As stated previously, depending on the current CPU state, the interpreter may either read 16 bits or 32 bits from memory which make up the instruction. Ideally, this read is performed through a delegation of different read-methods based on opcode size such that the only difference between these two paths lies in method selection. The program counter contains the address of the instruction read, appropriately aligned to either 2 or 4 bytes, and shall be incremented by the size of the current instruction type after execution[8].

All reads and writes have to go through the bus in order to be assigned to the correct subcomponent or memory range. Section 2.1.2 introduced the concept of MMIO which is responsible for handling the mapping between memory addresses and memory ranges. An implementation of such a system should seek to optimize this delegation of ranges as much as possible — guiding the compiler towards the generation of a jump table if viable — as it will take up a big part of the interpretive overhead. Most commonly, an approach akin to how a page table operates is used, as seen in Listing 2.3. A page table stores address translations between virtual pages, that is, a fixed size unit of the address space, and physical memory addresses [Arpaci-Dusseau and Arpaci-Dusseau, 2023]. Thus, in order to facilitate the use of paging, the first 8 bits of each memory range may be used as a page number allowing faster translation between the full address and its delegated component.

As for the program counter, most software will only ever execute code from either cartridge ROM in which the code is stored, IWRAM after copying code over from ROM or inside of the BIOS for system calls. Still, in theory, code may be executed from anywhere within memory.

[8]This may depend on individual implementation. Some emulators increment PC before execution.

```rust
match address {
  0x0000..0x4000 =>
    /* Handle BIOS */,
  0x0200_0000..0x0300_0000 =>
    /* Handle IWRAM */,
  0x0300_0000..0x0400_0000 =>
    /* Handle EWRAM */,
  ...
}
```

(a) Naive MMIO implementation.

```rust
match address >> 24 {
  0x00 if address < 0x4000 =>
    /* Handle BIOS */,
  0x02 =>
    /* Handle IWRAM */,
  0x03 =>
    /* Handle EWRAM */,
  ...
}
```

(b) "Page Table"-like MMIO implementation.

Listing 2.3.: Guiding the compiler towards jump table generation by emulating page table behaviour. Less overhead for verifying if ranges contain `address` since the page number can be used as an offset into a list of branches [Kulot, 2023].

Decode. When given the raw binary opcode data from the previous step, both the actual CPU and its emulated counterpart must identify and decode the data so as to dispatch the correct instruction based on certain bit patterns. Most instruction set architectures are designed in such a way that families of instructions follow those patterns, for example, given an 8-bit load instruction, the upper 4 bits might identify the load-family whereas the lower 4 bits might store a source and destination register within 2 bits each.

ARM instruction encodings store a considerable amount of information in individual bits or bit ranges which influence the precise behaviour of the instruction. As explained in Section 2.1.1 and Figure 2.4, a single bit may modify the operation itself, whether to use an immediate or a register value, whether to set flags, etc. Treating these variations as dynamic conditions only evaluated at runtime would lead to significant overhead each time the instruction is executed. Rather, it is desirable to extract the influential bits during decoding and statically dispatch a special version of the instruction implementation based on these constants. This can be achieved via compile time evaluation, usually realized by abstracting the function signatures over generic parameters; referred to as *Non-Type Template Parameters* in C++ or *Const Generics* in Rust.

Static dispatch as an approach to turn "generic [instruction] code into specific code by filling in the concrete types that are used when compiled" [Klabnik and Nichols, 2022, see *Performance of Code Using Generics*] relies on all possible function signatures being known ahead-of-time for it to work; only compile-time constants may be used to fill in concrete values. As such, it is recommended to generate a table of all possible function permutations at compile-time, specifically an array of function pointers with every important bit either set or cleared. This is called a look-up table (LUT) and its size depends on the instruction type. It has been shown that an ARM instruction can be uniquely identified with just 12 bits, leading to 2^{12} or 4096 different permutations. THUMB instructions can be uniquely identified by their upper half, leading to 2^8 or 256 permutations. The LUT is then filled with the corresponding function pointers, based on the identifying bit range, and the generic parameters are substituted with their concrete values.

Finally, the same 12-bit or 8-bit pattern may be used as an index into the LUT, fetching the function pointer which can then be invoked immediately. A notable downside to this approach is the resulting "code bloat", that is, a rather significant increase in binary size since the compiler generates a unique function for each permutation.

Execute. The last step in the emulation loop consists of handling control over to the just decoded function responsible for recreating the instruction behaviour at a high-level. Though, as the ARM part of Figure 2.8 and Section 2.1.1 suggests, ARM instructions are subject to conditional execution. Therefore, if the T-Bit is cleared, an additional check must happen based on the uppermost four bits of the encoding and the current CPSR state. If the condition is met, or THUMB state is active, the execution step begins and the emulated function is called.

This process continues on indefinitely until either the CPU is halted or the emulator is forcefully terminated, incrementing the program counter each iteration and repeating the loop. Most applications will usually perform a branch as their final instruction to create an infinite loop when they are done or fall back to some other waiting routine, hence no termination is expected from the ROM unless an error has occured.

2.2.3. Just-in-Time Compilation (JIT)

Just-in-time compilation represents a different approach to machine code translation, both in programming language development as well as emulation development. It intents to mitigate the overhead of interpretation and repeat execution by directly turning certain code paths into host instructions, and executing said instructions, while the program is running — gaining the benefits of both static compilation and interpretation [Aycock, 2003].

[Deutsch and Schiffman, 1984] were one the first to use this implementation technique for their Smalltalk-80 system. Not only did they utilize dynamic translation between "v-code" (guest code) and "n-code" (native code) but they used caching to store the translated code to accelerate the code generation process. This enabled Deutsch and Schiffman to speed up the procedure invocation process over 90% of the time in the best case. Caching, generally, refers to keeping important data in a fast-to-access temporary memory location. In the

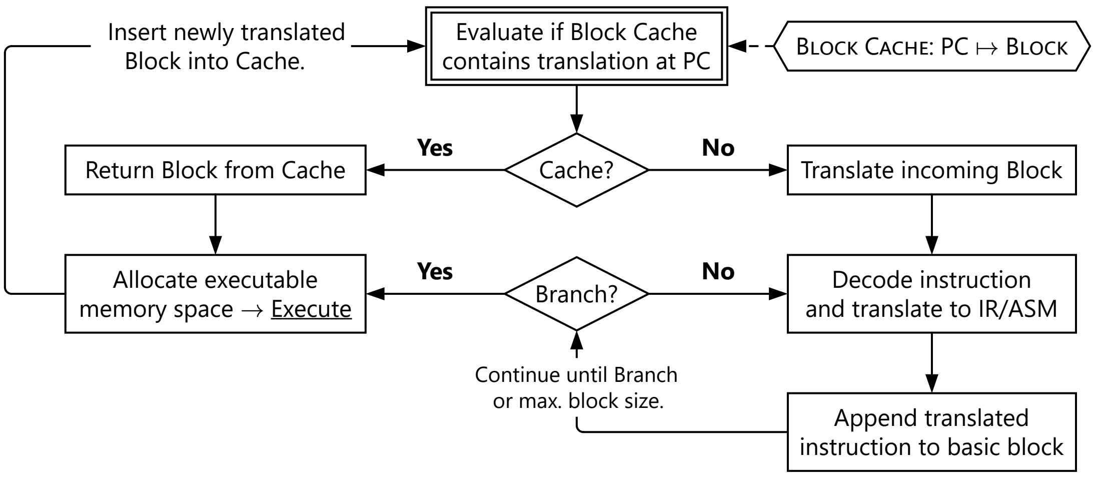

Insert newly translated Block into Cache.
Evaluate if Block Cache contains translation at PC
Block Cache: PC ↦ Block
Yes
No
Return Block from Cache
Cache?
Translate incoming Block
Yes
No
Allocate executable memory space → Execute
Branch?
Decode instruction and translate to IR/ASM
Continue until Branch or max. block size.
Append translated instruction to basic block

Figure 2.9.: JIT Emulation Loop. Logical extension of cached interpretation.

context of just-in-time compilation, it refers to storing sequences of translated instructions in-memory, such that no retranslation is necessary. As loops or branches appear quite often in machine code, certain instruction sequences tend to be executed repeatedly, leading to unnecessary overhead if caching is not used.

Figure 2.9 demonstrates this new process. Instead of looking at each instruction individually, they are instead collected into a sequence, or basic block, of instructions. A basic block refers to a sequence of branch-free code, beginning with a label and ending with an exit branch, which always executes together [Cooper and Torczon, 2022, p. 170]. Once a basic block is found, it then gets translated into host machine code and inserted into the cache with the value of the program counter at the beginning of the block as its key. If the key already exists within the cache, the translation process can be skipped entirely since the block has already been encountered. Executing in-memory code, then, requires memory-mapping to allocate a piece of memory with the permission to treat the data within as code. Lastly, this whole process repeats for and directly operates on each basic block; hence this type of JIT is often classified as *block-level*. Other types include *function-level* and *tracing* JITs.

Besides increased execution speed through native code generation, adaptive optimizations may be used allowing for more fine-tuned code based on the platform architecture. However, JITs can cause slight delays at startup during the initial phase of execution as nothing has been cached so far and code still has to be compiled. This may also happen when a completely new section of code is encountered, for example, when entering a different world in a video game. Thus, the more optimizations are applied during compilation, the more likely it is to notice startup delay or stutter — however, once that phase has concluded, there should be a notable increase in performance. Unfortunately, while ahead-of-time (AOT) compilation would in theory mitigate some of these issues, it is unfeasible to practically implement in the case of emulation. Most software uses self-modifying code which involves overwriting existing instructions at runtime or even just loading code from ROM into RAM at an unspecified, and not statically known, point in time. Furthermore, for systems with so-called "flat binaries", which mix data and code, it has been proven that detecting the separation of instructions from data is equivalent to the Halting Problem [Horspool and Marovac, 1978]. Ultimately, designing a JIT is a trade-off between generating optimal code and generating code at optimal speed.

2.3. LLVM

The LLVM Project [Lattner, 2002] is a set of modular compiler and toolchain technologies, designed around SSA-based (Static Single-Assignment) compilation strategies for both static and dynamic translation of programming languages. LLVM, originally, began as a research project at the University of Illinois but has quickly grown into one of the most widely used toolchains for compiler frontends, optimization and code generation as well as other novel approaches within the compiler space. Its core libraries are built around a code representation form known as LLVM IR; serving as a portable high-level assembly language and forming the basis for renowned compiler frontends such as `clang`[9], `rustc`[10] or `zig`[11].

[9] https://clang.llvm.org/
[10] https://www.rust-lang.org/
[11] https://ziglang.org/

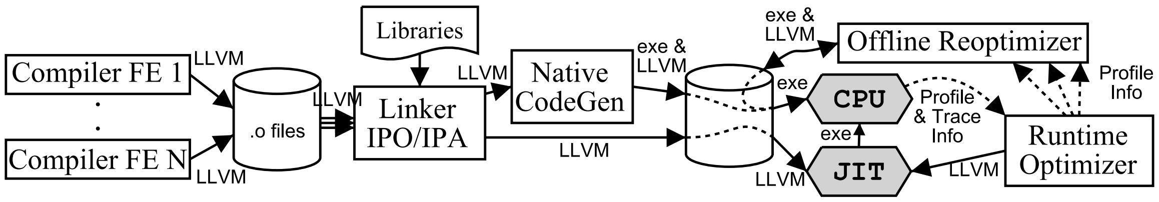


Figure 2.10.: LLVM system architecture diagram [Lattner and Adve, 2004].

One of the key benefits of using LLVM is its support for a wide range of architectures out of the box. Additionally, once source code has been translated into LLVM IR, a plethora of built-in optimizations can be applied resulting in more optimal execution and code generation. A notable downside of striving for optimal execution is the rather significant increase in compile time, or startup delay, as code is being generated [Xu and Kjolstad, 2021, p. 3]. Figure 2.10 shows an abridged version of this process, starting with frontends using LLVM to turn their source into LLVM IR before native code generation takes place. Following that, more optimization passes get applied and the final code is either stored in an executable or in-memory for possible just-in-time execution.

2.4. Cranelift

Cranelift is a fast and secure compiler backend, intended for use as a library as part of an overarching toolchain [Bytecode Alliance, 2024]. Analogous to LLVM, Cranelift also uses an intermediate representation, targets multiple platforms and can be used for both AOT and JIT compilation. The main differences stem from its approach towards code generation. Unlike LLVM, only one form of IR is used throughout the entire process, its SSA-form is preserved throughout and no mid-level optimizations are applied. According to [Gohman, 2018], "this biases the overall system towards fast compilation [...] when emitting unoptimized code for or when low-level optimizations are sufficient". As such, Cranelift offers a potential solution in terms of the trade-off between lower startup delay and less optimal code.

The Wasmtime[12] WebAssembly virtual machine uses Cranelift for JIT and AOT generation of WASM code as it was originally developed to be a WebAssembly compiler for the SpiderMonkey[13] engine within Mozilla Firefox. It also serves as an experimental debug backend for the Rust compiler with the explicit goal to reduce compilation times. Nonetheless, Cranelift aims to be a fully general and retargetable compiler backend with strong focusses on correctness [Fallin, 2021], verification [Clark and Schneidereit, 2020; VanHattum et al., 2024] and innovative approaches [Cabrera-Arteaga et al., 2024; Fallin, 2022a] in the field.

[12]`https://github.com/bytecodealliance/wasmtime`

[13]`https://github.com/Amanieu/cretonne/blob/master/spidermonkey.rst#phase-1-webassembly`

Chapter 3.

Design and Implementation

In this chapter, a few of the most important design choices of each compiler framework regarding usability, fast execution, fast code generation and overall relevance for implementing emulation will be explored. This includes the respective IRs and how well they map to ARMv4T, how register allocation affects both compilation time and code quality, how control flow is realized in an SSA environment and which optimizations can be included without compromising on the just-in-time aspect. Implementation examples will be shown where applicable.

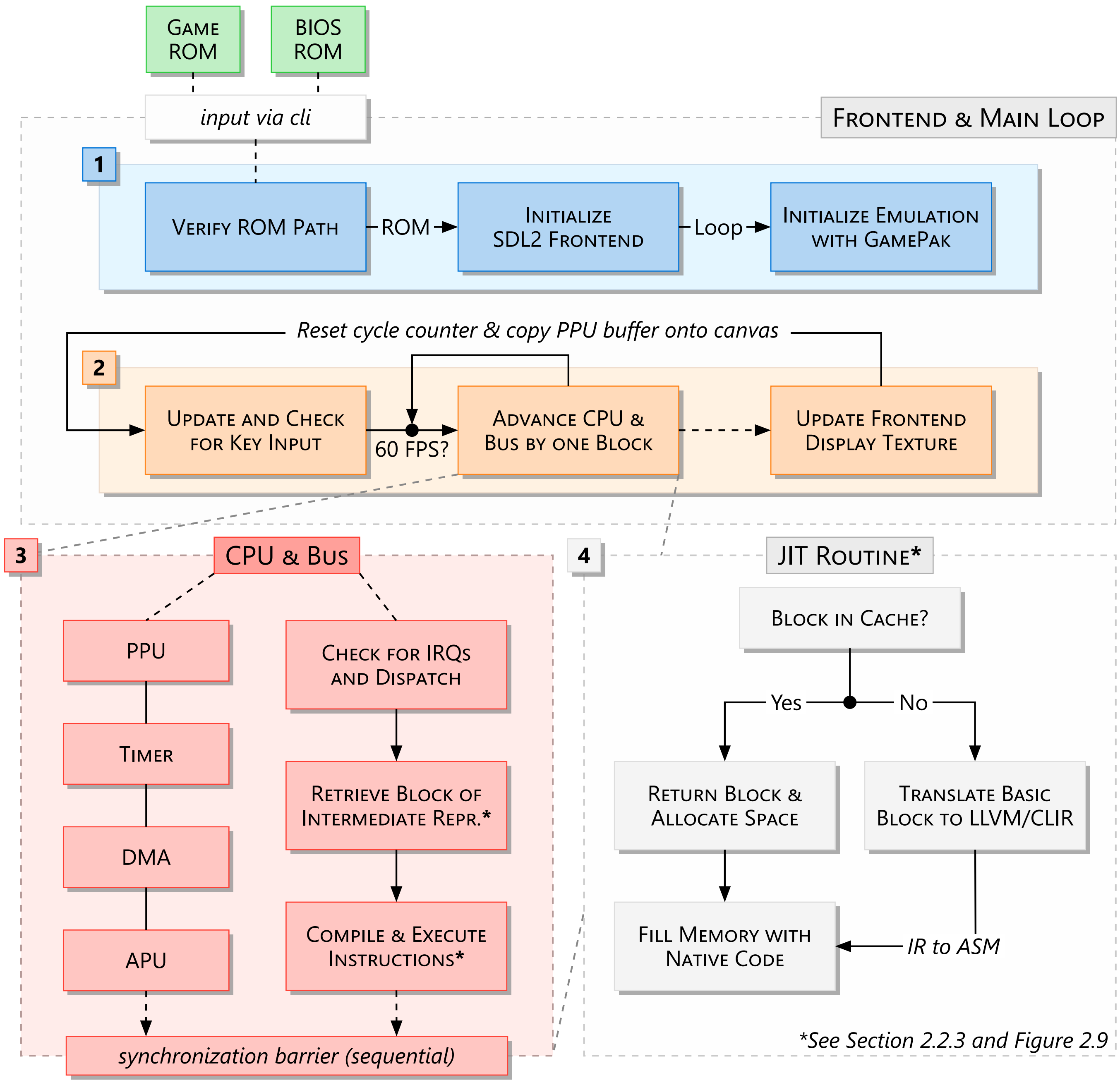
3.1. Structural Emulation Overview

The design and implementation part of this thesis required thorough research into the behavior and functionality of the Game Boy Advance system and the creation of an emulator with three different CPU cores, namely the interpreted variant, the LLVM JIT and the Cranelift JIT. A modular design was chosen such that every part of the system can be reused and only the CPU implementation has to be replaced. In principle, the CPU interface should operate in an independent fashion irrespective of the greater system it was placed in. This decoupling of the processing unit, system emulation and the graphical frontend enables not only a straightforward extension of additional subsystems but also removes certain complexities when debugging since no inter-subsystem dependencies entangle the system architecture.

Figure 3.1 shows the corresponding system architecture diagram. First, both a game ROM and a BIOS ROM need to be provided by the user via its command-line interface. Following verification of the file paths and proper initialization of the frontend, both ROMs are used to instantiate the `Gba` struct which holds the `Arm7TDMI` struct, the `Bus` struct, an internal cycle counter and the ROM data inside a byte vector. While the BIOS startup screen (described in Section 2.1.2) and initialization routine can, in theory, be skipped by replacing internal register values with known post-BIOS values, its presence inside the memory space is still necessary as software can and will use software interrupts to call aforementioned BIOS routines.

SDL2[14] (Simple DirectMedia Layer), a cross-platform development library providing low-level access to graphics hardware and more, is used for the graphical user interface; displaying the screen texture and handling key input. After the window is created, an event pump, a canvas and a texture creator are stored in a shared `SDLApplication` struct used for later access in the main update loop. Just before the main loop is entered, the texture creator creates a "streaming" texture, that is, a special kind of texture intended to be updated often and assuming full

[14] `https://www.libsdl.org/`



Game ROM
BIOS ROM
input via cli
Frontend & Main Loop
1
Verify ROM Path
ROM
Initialize SDL2 Frontend
Loop
Initialize Emulation with GamePak
Reset cycle counter & copy PPU buffer onto canvas
2
Update and Check for Key Input
60 FPS?
Advance CPU & Bus by one Block
Update Frontend Display Texture
3
CPU & Bus
PPU
Timer
DMA
APU
Check for IRQs and Dispatch
Retrieve Block of Intermediate Repr.*
Compile & Execute Instructions*
synchronization barrier (sequential)
4
JIT Routine*
Block in Cache?
Yes
No
Return Block & Allocate Space
Translate Basic Block to LLVM/CLIR
Fill Memory with Native Code
IR to ASM
*See Section 2.2.3 and Figure 2.9

Figure 3.1.: Structural architecture of the Game Boy Advance emulation implementation. From reading in the ROMs to compilation and execution.

byte array updates from external source data [Lantinga, 1997], hence ideal for the use case of *blitting* the internal PPU buffer into the frontend buffer every frame. Next, the main loop begins and runs indefinitely until forceful termination; it queries and polls the event pump which only handles the `Quit` event breaking out of the loop if necessary, a macro handles the check for key presses — passing the result on to the internal key input registers — and the emulator is instructed to step up to n instructions per `run()` call (one when interpreted, a basic block when a JIT is used) until the logical limit per frame is reached. However many instructions the Game Boy Advance is able to execute within the time frame of 1/60th of a second at 16.78 MHz, the emulator tries to achieve as well. As such, the texture creator is instructed to update its managed texture after roughly $\frac{16.78\text{ MHz}}{60\text{ FPS}} \approx 279.666$ cycles, clearing the canvas, copying the texture onto it and presenting its backbuffer to the screen.

As the emulator is stepped n cycles, both the CPU and the rest of the system are updated in tandem, as seen in block 3 of Figure 3.1. Unless the CPU is halted, the interrupt registers are checked for pending requests and dispatched and the CPU advances by one cycle / one basic block. In the meantime, the `Bus` also advances by the appropriate number of steps along with all its subcomponents. Listing 3.1 consists of the code written for the jitted CPU `run()`-method: Any code inside the second if-condition on line 7 corresponds to the three nodes on the right-hand side of block 3 in Figure 3.1, lines 17–18 correspond to the two lower ones marked with an asterisk and implement the common JIT workflow first presented in Figure 2.9 (block 4). Details depend on the framework. Furthermore, the progression of the nodes on the left-hand side (PPU, Timer, DMA, APU) corresponds to line 25 which is further explored in Listing 3.2. Lastly, the internal cycle counter receives the amount of cycles that have passed within the last block as the `Arm7TDMI::get_next_block()`-method returns both a `Block` and its cycle count. In order to keep the emulation simple and performant, a cycle rate of 1 cycle per instruction (CPI) is assumed, thus the cycle count per block is equal to the amount of instructions per block.

```rust
 1  pub fn run(&mut self, jit_translator: &mut JitTranslator) {
 2      if self.cpu.bus.halt && (self.cpu.bus.ie.0 & self.cpu.bus.iff.0) != 0 {
 3          self.cpu.bus.halt = false;
 4      }
 5
 6      let block_cycles =
 7        if !self.cpu.bus.halt {
 8            self.cpu.dispatch_irq();
 9
10            /*
11            Either, retrieve block from cache if already seen and compiled,
12            or translate into IR/ASM until branch w/ either LLVM or Cranelift.
13
14            Then, allocate executable memory buffer and fill it with the just
15            compiled native machine code. Then, transmute into function.
16            */
17            let (block, cycles) = self.cpu.get_next_block(jit_translator);
18            self.cpu.allocate_and_execute_block(block);
19
20            cycles
21        } else {
22            1
23        }
24
25      for i in 0..block_cycles { self.cpu.bus.tick(self.cycles + i) }
26      self.cycles += block_cycles;
27  }
```

Listing 3.1.: A single JIT step for both the CPU and the rest of the system.

```rust
/// Tick PPU --> Timers --> DMA by one cycle.
/// - All components receive a mutable reference to the IRQ register.
/// - Should their IRQ conditions apply, they can request it with `iff`.
/// - Timers receive current cycles to match their set frequencies.
pub fn tick(&mut self, cycles: usize) {
    // Cycle the PPU state machine.
    self.ppu.cycle(
        &*self.vram,
        &self.palette_ram,
        &self.oam,
        &mut self.iff,
    );

    // Tick all four timers simultaneously.
    self.timers.tick(&mut self.iff, cycles);

    /*
    The following DMA checks can still be optimized if they are only called
    directly when HBlank or VBlank happens, instead this still checks stuff
    every cycle but doesn't run it every cycle.

    Similar for Immediate DMA. Problem is getting `self.dma_transfer` from
    the borrow-checker into the PPU state machine.
    */

    // On state/mode change.
    if self.ppu.prev_mode != self.ppu.current_mode {
        use crate::ppu::lcd::Mode;
        match self.ppu.current_mode {
            Mode::HBlank => self.dma_transfer(StartTiming::HBlank),
            Mode::VBlank => self.dma_transfer(StartTiming::VBlank),
            Mode::HDraw => {},
        }

        self.ppu.prev_mode = self.ppu.current_mode;
    }

    // On enable transition for immediate DMAs.
    if (0..4).any(|ch| self.dma_channels[ch].enable_edge()) {
        self.dma_transfer(StartTiming::Immediate);
    }
}
```

Listing 3.2.: A single step for the Bus and its subcomponents. As the Audio Processing Unit was not implemented to save time and resources, it is not included here.

Real hardware would, of course, not cycle through each component sequentially but rather each circuit would run in parallel or in a concurrent manner. However, this is not possible with pure software emulation without the use of multi-threading which introduces a completely different problem space with regards to synchronization. As the sequential approach does not affect accuracy in most software — provided that the console state visible to software is as expected — having each component tick or cycle one after another will suffice. The realization of this generalized sequence can be seen in Listing 3.2: One after another, each subcomponent advances their internal state by one step. First, the PPU enters its state machine responsible for managing the current mode and upcoming events such as rendering a scanline, increasing the LY register or setting other internal flags to request certain interrupts. Next, the `Timers` tuple struct, which essentially only consists of an array with four individual `Timer` structs, ensures that only enabled timers tick up when their condition is met. DMAs are managed differently depending on their type. Non-immediate DMAs check for PPU mode changes and, if the mode matches their set `StartTiming`, the transfer begins whereas immediate DMAs begin their transfer as soon as they detect a rising edge for the enable bit.

Ultimately, having a tick-based update architecture for IRQs and bus-managed subcomponents represents just one approach towards implementing the behaviour outlined in Section 2.1.2. A major downside of constant component cycling is that for the majority of the time, nothing actually needs to be done and, therefore, every conditional check incurs unnecessary overhead. Section 6.2 briefly introduces a scheduler-based approach which intents to mitigate the issue of excessive component management with the use of timestamped events.

3.2. Intermediate Representation

Compilation undergoes many intricate stages before emitting the target assembly language. One of those is the transformation to and subsequent use of an *intermediate representation* (IR) which lies somewhere between an abstract form of the source language and the concrete structure of native instructions [Thain, 2020]. Besides creating a higher level of abstraction and facilitating platform independence, as briefly mentioned in Section 1.1, an IR should be designed to be of simple and regular structure as to assist optimization, analysis and efficient code generation. Complex compilation pipelines may use multiple IRs with decreasing levels of abstraction.

Different types of IRs can be used depending on the level of optimization and introspection needed; Cranelift and LLVM both expose an API to programmatically generate their respective IRs in the so-called Static Single Assignment Form (SSA). Other types include the Abstract Syntax Tree (AST) itself, a Directed Acyclic Graph (DAG) of value and instruction nodes, a Control Flow Graph (CFG) which is a directed graph consisting of basic blocks with edges representing the control flow between them or virtual stack machines. These can either be defined in an external format, that is, a textual version of the IR or in-memory via usage of the appropriate data structures provided by the API.

For instance, Listing 3.3a demonstrates how to use the Cranelift API to generate the appropriate intermediate representation for a simple function which adds one to its incoming parameter and returns the result. Listing 3.3b, on the other hand, uses a textual form to describe this function which, while syntactically different, should result in identical semantics. Both will compile to the same native machine code, though the textual form is usually reserved for testing and debugging purposes as it is more human-readable than *builder patterns*.

The following sections will go into more detail for each step, explaining SSA, why it is used and the differences between the LLVM and Cranelift code generation API, including a proper analysis of the code in Listing 3.3 once more concepts have been introduced.

```rust
// Create a new block and return its reference.
let block = builder.create_block();

// Declares that all the predecessors of this block are known.
builder.seal_block(block);

/*
  Append parameters to the given Block corresponding
  to the function parameters. Then, tell builder to emit
  IR into the just-created block.
*/
builder.append_block_params_for_function_params(block);
builder.switch_to_block(block);

// In-Memory representation of CLIR:
// - Get passed block parameter `v0`.
// - Add 1 to `v0` and store in `v1`.
// - Return `v1`.
let arg = builder.block_params(block)[0];
let plus_one = builder.ins().iadd_imm(arg, 1);
builder.ins().return_(&[plus_one]);

builder.finalize();
```

(a) In-Memory representation of Cranelift IR, generated via its `FunctionBuilder` interface.

```
;; Textual form of CLIR.
function u0:0(i64) -> i64 system_v {
block0(v0: i64):
    v1 = iadd_imm v0, 1
    return v1
}
```

(b) Textual representation of Cranelift IR for the same code; usually stored within `.clif` files.

Listing 3.3.: Comparing different forms of the same representations for equivalent semantics. The textual form was generated from the `Block` data structure via the `display()`-method of a `Function` object[15] [de Moraes, 2022].

[15] `https://docs.rs/cranelift-codegen/latest/cranelift_codegen/ir/function/struct.Function.html#method.display`

Static Single Assignment Form (SSA)

Commonly used for more complex optimizations, the static single assignment form is a form of intermediate representation with one important restriction for each basic block: all variables may only be assigned to exactly once, existing variables are split into versions and receive a new version number when assigned a new value [Rosen et al., 1988; Thain, 2020, p. 127]. Its primary usefulness stems from a simultaneous simplification of compiler optimizations as well as improving their results. For instance, a human can easily see that there is no need to create both a variable called x and a just to store the value of one at line 1–2 of Listing 3.4a, whereas the compiler would have to perform a *reaching definition analysis* to apply *constant propagation* to substitute the values at compile time.

```
1  let x = 1;
2  let a = x;
3
4  let b = a + 10;
5  x = 20 * b;
6  x = x + 30;
```

(a) Pseudo code.

```
1  x_1 := 1;
2  a_1 := x_1;
3
4  b_1 := a_1 + 10;
5  x_2 := 20 * b_1;
6  x_3 := x_2 + 30;
```

(b) SSA-Form.

```
1  v1 = iconst.i32 1
2  v2 = v1
3
4  v3 = iadd_imm v2, 10
5  v4 = imul_imm v3, 20
6  v5 = iadd_imm v4, 30
```

(c) Cranelift IR.

Listing 3.4.: Version numbering and single assignment for a hypothetical SSA IR and valid Cranelift IR in textual form, based on [Thain, 2020, p. 127].

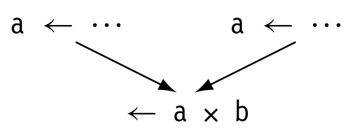
Now, with Listing 3.4b and Listing 3.4c, the compiler will still propagate known constants at compile time but it is much clearer where the usages of x and a begin and end as they cannot be redefined. In a similar fashion, dead store and dead load elimination become trivial. Assignments such as `y := 1; y := 2; x := y;` would, once again, require a reaching definition analysis to determine the unnecessary first assignment. Conversely, its static single assignment representation `y_1 := 1; y_2 := 2, x := y_2;` makes it immediately clear that y_1 represents not only a dead load but also dead code and can therefore be optimized out.

So far, assignments have been straightforward but when a variable is given different values inside the branches of a conditional, it becomes unclear how this would be realized with the given rules of SSA. It is not statically known which branch will be taken and, thus, the variable could have either value which is prohibited as per the single assignment rule. To resolve this ambiguity, a new function $\phi(x, y)$ is introduced indicating that either x or y could be selected at runtime. This preserves the control flow and links the new value to all possible old values; [Cytron et al., 1991] were one of the first to propose new algorithms for efficient computation of control flow graphs in SSA form which involves inserting ϕ-functions at the program's join nodes. A visualization of this graph transformation can be seen in Figure 3.2: The generic CFG (Figure 3.2b) is able to use the value of a regardless of previous branches. After translation to SSA form, the expression $a_2 \leftarrow \phi(a_0, a_1)$ has to be inserted at the join node just before the multiplication (Figure 3.2c) as to correctly assign the old value of a_0 or a_1 to a_2 while adhering to the aforementioned restrictions.

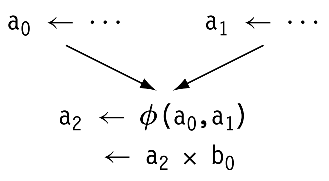
It is left up to the compiler how to realize this when translating to assembly. Most will implement some kind of copy-routine through conditional move-instructions for example, though this is highly dependent on prior optimization passes.

```
let a = cond ? 1 : 2;
let b = 5;
let c = a * b;
```

(a) Pseudo code.



(b) Control Flow Graph.



(c) SSA Form with ϕ.

Figure 3.2.: Depending on how control flow entered the block, the ϕ-function evaluates a different argument. <cond> represents an arbitrary condition assessed via the ternary operator [Cooper and Torczon, 2022, p. 469].

LLVM IR

LLVM features multiple intermediate representations as it translates a program from source to machine code. First, LLVM IR, which is the primary intermediate representation, aims to be a "universal IR", designed to be "light-weight" and "low-level while being expressive and typed" according to [Lattner and Adve, 2012]. It is designed to be used in three different forms: an in-memory representation, an on-disk bitcode representation and a human readable textual form. As a large library of common mid-level optimizations, code analysis and transformation passes operate on LLVM IR, this project also uses LLVM IR to emulate the instrution set.

Further IRs include the SelectionDAG, representing a graph-based representation of the code within a single basic block. Both ISA-agnostic and ISA-specific opcodes are used in the DAG, the instruction selector then makes use of this information. The following passes are run on the SelectionDAG representation [Gohman, 2018]:

• **Type Legalization:**	Eliminates value types without representation in ISA.
• **Operation Legalization:**	Eliminates opcodes that cannot be mapped to target ISA.
• **DAG-Combine:**	Cleans up redundant code after the legalization passes.
• **Instruction Selection:**	ISA-agnostic expressions to ISA-specific instructions.

MachineInstr and MC are the final layers in the LLVM representation transformation pipeline. Scheduling and register allocation happen in the MachineInstr phase — an extremely abstract way of representing machine instructions. MC serves as the basis for LLVM's integrated assembler and is used to represent code at the raw machine code level. Assembling, disassembling, branch relaxation and emitting object code happens during this phase.

LLVM IR syntax is shaped by its SSA characteristics and follows an assembly-like structure for operand order. Identifiers can either be global and are prefixed with the `@`-character, mostly for functions and global variables, while local identifiers begin with the `%`-character. Functions must be declared with the `declare` keyword and defined with the `define` keyword and consist of a list of basic blocks forming the function's CFG. A full example of a working LLVM IR program including functions, instructions and constant declaration can be seen in Listing 3.5: Line 1 declares and defines a global constant with the `@`-prefix and the `constant` keyword containing the string `"hello world\n\0"`. Line 5 *declares* the `puts` function such that the linker knows to insert or jump to the corresponding function definition when it is called. The `main` function is *defined* in line 8 and includes two instructions showcasing one variant of possible operand orders. Instructions without results, like `call` or `ret`, follow a `<op> <type> <arg1>` structure whereas instructions with results (`add`, `shl`, `mul`, etc.) would be called like `<dst> = <op> <type> <arg1> <arg2>` instead. Parameter attributes communicate additional information and follow the specified type; LLVM defines a plethora of these attributes, for example `nocapture`, `nounwind` or `noalias` to name a few.

```
 1  ; Declare the string constant as a global constant.
 2  @.str = private unnamed_addr constant [13 x i8] c"hello world\0A\00"
 3
 4  ; External declaration of the puts function
 5  declare i32 @puts(ptr nocapture) nounwind
 6
 7  ; Definition of main function
 8  define i32 @main() {
 9    ; Call puts function to write out the string to stdout.
10    call i32 @puts(ptr @.str)
11    ret i32 0
12  }
```

Listing 3.5.: "Hello World" in LLVM IR, showcasing its high-level module structure [Lattner and Adve, 2012, see *Module Structure*].

For emulation and general compilation, it is preferable to dynamically generate an in-memory representation of the IR. Hence, the `inkwell` ("**I**t's a **N**ew **K**ind of **W**rapper for **E**xposing **LL**VM") [Kolsoi, 2023] crate is used as a safe wrapper around the Rust bindings to LLVM's C API to access the necessary data structures. Listing 3.6 features an abridged version of example Rust code using the LLVM API to generate a function, which — as before — takes one parameter as input, adds one to it and returns the result. Lines 2–12 in Listing 3.6a are all responsible for initializing and creating important types before the function definition can begin. This prelude consists of specifying the exact types used (Section 3.5), creating a proper function signature, declaring it within the module and preparing the instruction builder to append instructions at the correct position within the basic block. Finally, lines 14–18 actually specify which instructions to emit before the function is compiled.

A concrete implementation for the `MUL`/`MLA` opcode using LLVM is featured in Listing 3.7. The first listing shows how the interpreter imitates the high-level behavior of the instruction: Both the destination register and whether the accumulator should be used are extracted from the numeric value of the opcode itself, that is, bits 11–15 and bit 21 respectively. Bits 0–3, 4–7 and 8–11 specify the registers in which the first operand, the second operand and the optional third operand are stored. Then, operand 1 and operand 2 get multiplied together and, if the accumulator bit is set, the third operand is added onto the intermediate result. Finally, CPSR is optionally modified as the `N` and `Z` flag are adjusted as necessary depending on the `S`-bit. Listing 3.7b shows this exact same implementation but realized through LLVM's `Builder` API to generate the IR in-memory. Just as before, the relevant register values are extracted from the opcode — still with high-level code as those are known at compile-time — and some IR SSA variables are defined for later use in lines 17–18. Following that is a close 1:1 translation of the actual calculation; the first two operands are passed into `build_int_mul` which constructs the IR for the `mul` instruction and appends it to the function called `"mul"`. The `build_int_compare` method creates the `icmp` instruction which evaluates a comparison based on its parameters and the given predicate — this decides whether the accumulator is used. Based on this, `build_select` either returns the to-be-added value or zero and adds it, yielding the final result.

More thorough examples regarding the implementation of ARM7TDMI opcodes in LLVM IR and their subsequent code generation can be seen in Section 5.2.1 and Section 5.2.2 with their original interpreted implementation, the in-memory JIT implementation using the respective APIs, the generated textual IR and how it is transformed into native x86 code.

```rust
// Define function type as function that returns `i64` and takes one `i64`.
let i64_type = context.i64_type();
let fn_type = i64_type.fn_type(&[i64_type.into()], false);

// Add function to module and give it a name.
let function = module.add_function("add_one", fn_type, None);

// Create a new basic block and return its reference.
let basic_block = context.append_basic_block(function, "entry");

// Make sure builder appends instructions to block.
builder.position_at_end(basic_block);

let x = function.get_nth_param(0)?.into_int_value();
let one = i64_type.const_int(1, false);
let x_plus_one = builder.build_int_add(x, one, "add_one").unwrap();

builder.build_return(Some(&x_plus_one)).unwrap();
unsafe { execution_engine.get_function("add_one").ok() }
```

(a) In-Memory representation of LLVM IR.

```llvm
@one = private unnamed_addr constant i64 1

define i64 @add_one(i64 %x) {
  %one = load i64, i64* @one
  %x_plus_one = add i64 %x, %one
  ret i64 %x_plus_one
}
```

(b) Textual representation of LLVM IR.

Listing 3.6.: Comparing the LLVM API exposed through `inkwell` with its textual equivalent. This code defines the same function as in Listing 3.3.

```rust
pub fn multiply<const S: bool>(&mut self, opcode: u32) {
    let acc = (opcode & (1 << 21)) != 0;

    let rd = (opcode as usize & 0x000F_0000) >> 16;
    let rm = self.regs[opcode as usize & 0xF];
    let rs = self.regs[(opcode as usize & 0x0F00) >> 8];
    let rn = self.regs[(opcode as usize & 0xF000) >> 12];

    self.regs[rd] = rm * rs + (rn * acc as u32);

    if S {
        self.cpsr.set_n(self.regs[rd] & (1 << 31) != 0);
        self.cpsr.set_z(self.regs[rd] == 0)
    }
}
```

(a) Interpreter emulation. Accumulator boolean is used as an operand.

```rust
pub fn multiply<const S: bool>(
  &mut self,
  opcode: u32,
  jit: &mut JitTranslator
) {
  let builder = &mut jit.builder;
  let context = &mut jit.context;
  let i32_type = context.i32_type();

  let acc = (opcode & (1 << 21)) >> 21;
  let rd = (opcode as usize & 0x000F_0000) >> 16;
  let rm = jit.regs[opcode as usize & 0xF];
  let rs = jit.regs[(opcode as usize & 0x0F00) >> 8];
  let rn = jit.regs[(opcode as usize & 0xF000) >> 12];

  let zero = i32_type.const_zero();
  let rn_val = i32_type.const_int(rn, false);
  let acc_val = context.bool_type().const_int(acc, false);

  let rmrs = builder.build_int_mul(rm, rs, "mul")?;
  let is_acc = builder.build_int_compare(
    IntPredicate::NE, acc_val, zero, "mul")?;
  let acc_add = builder.build_select(is_acc, rn_val, zero, "mul")?;

  jit.regs[rd] = builder.build_int_add(rmrs, accadd, "mul")?;
}
```

(b) LLVM JIT. Accumulator is assessed via the `select` instruction.

Listing 3.7.: Original interpreter implementation of the `MUL/MLA` instructions compared with the LLVM implementation using its `Builder` API.

Cranelift IR (CLIR)

Cranelift uses just one intermediate representation to cover all levels of abstraction. This is in part due to the reasons outlined in Section 2.4 as well as Cranelift's smaller scope. As previously mentioned at the beginning of Section 3.2 and visible in Listing 3.3, CLIR has two primary forms: an in-memory data structure generated from the `cranelift-codegen` crate and a text format used for testing and debugging which uses the `.clif` file extension.

Instructions in Cranelift IR use and produce values in SSA form but unlike LLVM IR, there is no dedicated ϕ-instruction to specify conditional assignment. Instead, basic blocks can be defined with a list of typed parameters such that whenever control is transferred to this basic block, these argument values must be provided. Incoming function arguments are also automatically passed into the entry block. Execution can never fall through to the next basic block since every block must end with a *terminator instruction* (`return`, `call`, `trap`, etc.). The `cranelift_frontend` crate features some utility for translating multiple assignments into SSA form on its own when generating CLIR.

Looking at Listing 3.3 now, it becomes clearer how both representations work. Listing 3.3a has to perform certain Cranelift-specific method calls due to the aforementioned basic block parameter list: A block should be *sealed* once all blocks which could link, or jump, to it have been defined. Function parameters can be appended to block parameters and then accessed with `block_params()`, see line 19, and not every builder-method needs to be `unwrapped` or handled in some other way, leading to a cleaner API. Besides these differences and some code for creating the top-level function along with other initialization routines which were cut for the sake of brevity, the general IR creation through some form of `Builder` API remains similar. The `MUL`/`MLA` opcode implementation realized through Cranelift, for example, uses an almost identical IR structure as its LLVM counterpart in Listing 3.7 with the following CLIR instructions: `imul`, `icmp_imm`, `select` and `iadd`.

Unlike LLVM IR, however, its textual form uses a much simpler grammar (Listing 3.8) without the need for `declare`, `define` or any of the other keywords. Functions are self-contained, allowing for independent compilation, and requiring no explicit module structure which has to contain the functions. Listing 3.3b, and to an extent Listing 3.4c, demonstrate this simplicity by showcasing a more modern syntax. For instance, trailing return types use the arrow syntax and parameter types are declared after the parameter name. Additionally, as there is no need to differentiate between local and global variables, no special prefixes are used. Rather, each variable must follow the SSA rule of increasing version numbering which is explicitely enforced through their naming convention of `vX` where `X` is the version number. Stack slot values follow the `ssX` naming convention.

```
function_list : { function }
function      : "function" fn_name signature "{" preamble function_body "}"
preamble      : { preamble_decl }
function_body : { basic_block }
```

Listing 3.8.: Top-level grammar for textual CLIR. A `.clif` file consists of a sequence of function definitions [bjorn3 and Singh, 2020]. Rest of grammar at Listing A.1.

3.3. Instruction Block Caching

In order to enable the main advantage of just-in-time compilation, the translated machine code is cached to mitigate recompilation overhead. As previously stated in Section 2.2.3, certain code sequences tend to be executed repeatedly as high-level constructs such as loops and function calls are compiled down to conditional `jumps` and `calls` — some sequences might repeat thousands of times. Thus, storing those sequences and having them ready for dispatch as soon as the program counter reaches the beginning of the block, improves not only compilation time as a whole but should provide near-native performance in already cached areas as well.

The cache implementation is done via a naive key-value storage, that is, a `HashMap` mapping from the unsigned 32-bit program counter value at the beginning of the to-be-cached instruction sequence to a tuple of the native code as a byte vector together with its corresponding ARMv4T cycle count. As the JIT is single-stepped, the call to `cpu::get_next_block` (Listing 3.1) takes a `jit_translator` as a parameter which is, among other things, responsible for managing the cache and early-returning from said call whenever a cache hit occurs. Besides cache management, the `JitTranslator`, outlined in Listing 3.9, is responsible for keeping track of all sixteen general purpose registers in SSA form, being able to reference the top-level module to access externally declared functions and storing the IR builder hence why a mutable reference to this struct is always passed into the opcode implementation methods.

```rust
/// JIT Translator, builds and generates IR.
pub struct JitTranslator<'ctx> {
    /// Function builder to build functions and instructions.
    builder: FunctionBuilder<'ctx>,
    /// Cache instruction blocks based on the program counter (r15).
    blocks: HashMap<u32, (Vec<u8>, usize)>,
    /// Refer to the JIT Module to access and declare data/funcs.
    module: &'ctx mut JITModule,
    /// SSA representation of GPRs.
    regs: [Variable; 16],
}
```

Listing 3.9.: Caching is managed through the `JitTranslator` struct. The cache is implemented with a `HashMap` which uses the program counter as the key and a tuple representing the instruction bytecode and its cycle count as a value.

3.4. Register Allocation

Register allocation describes the problem of mapping a program P_v with an unbound number of virtual registers to a program P_p with a limited number of physical registers. During code generation, the compiler first uses virtual registers and assumes a sufficient amount of physical registers in the target machine. Generally, the first compilation passes will require more physical registers than the hardware provides, hence a register allocator must perform the aforementioned mapping. The register allocator determines which values will reside in registers and which will reside in memory — typically through rewriting code by adding load and store instructions to move values around between registers and memory [Cooper and Torczon, 2022, pp. 663–664] — ensuring optimal register usage for improved runtime performance. Moving a value from register to memory is called a "spill"; retrieving a previously spilled value is called a "restore". A superficial example of a simple register allocation can be seen in Listing 3.10, which demonstrates both a spill and a reload on a hypothetical machine with just two physical registers. Line 2 shows the first spill onto the stack, with sp as the stack pointer, as the virtual add instruction uses three virtual registers whereas the physical machine performs the addition with just two registers, overwriting r1 to store the result. Later, on line 6, the previously spilled value is restored and put in r0 for use as a multiplication operand with the original value of v1.

```
 1  @ Virtual-register code.
 2  @ Three live values -> Spill.
 3  add v2, v0, v1
 4
 5  sub v3, v2, v0
 6
 7  @ "Two" live values -> Reload.
 8  mul v4, v3, v1
 9
10  str v4, [sp + 48]
```

(a) Virtual-register code.

```
 1  @ v0 -> r0, v1 -> r1
 2  str r1, [sp + 0] ; *SPILL*
 3  add r1, r0, r1
 4  @ v0 -> r0, v1 -> [sp], v2 -> r1
 5  sub r1, r1, r0
 6  ldr r0, [sp + 0] ; *RELOAD*
 7  @ v1 -> r0, v3 -> r1
 8  mul r0, r1, r0
 9  @ v4 -> r0
10  str r0, [sp + 48]
```

(b) Register-allocation result.

Listing 3.10.: Example of a simple register allocation problem with the allocation performed onto a machine with two *real* registers (r0, r1) [Fallin, 2021].

Data movement between registers and memory and among registers should be minimized to improve code efficiency. Specifically, this means that "if the allocator decides to keep some value x in a physical register, it should arrange, if possible, for each definition of x to target the same [physical register] and for each use of x to read that [physical register]" [Cooper and Torczon, 2022, p. 666], eliminating redundant register-to-register copy operations. In order to track the life time of a value and its needed storage, so-called "live ranges" are used which connect a value in a virtual register from the initial definition up to its uses. Cooper and Torczon define a live range (LR) as a "closed set of related definitions and uses" which "most allocators use [...] as values that they consider for placement in a physical register or memory". The longer the live range, the more likely it is for the compiler to store a value inside a physical register.

Figure 3.3 shows a conversion from a custom IR to live range values with a corresponding graph of LR intervals aligned to their start and end. In order to determine if two LRs can occupy the same register, the compiler must visit each operation node and add possible interferences, that is, two LRs using the same class of registers but an operation takes place where both are live, to create an *interference graph*. An interference graph is an undirected graph $G = (N, E)$ that has a node n for each LR and an edge $(\mathrm{LR}_i, \mathrm{LR}_j)$ if and only if LR_i and LR_j interfere.

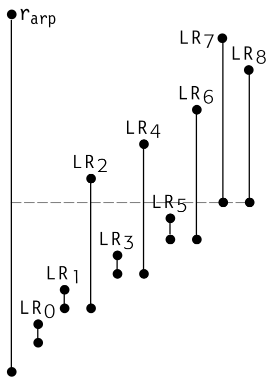
The degree of a node LR_i in G, written as LR_i°, represents the amount of neighbors the node has and is often used as an estimate for register demand or register pressure, meaning the number of simultaneously live variables at an instruction [Braun and Hack, 2009]. It is ultimately up to the allocator to determine optimal spill and reload locations, detect interferences and how to represent registers; at its core it represents a combinatorial optimization problem with NP-complete subprocesses underpinning all decisions. Graph coloring, linear scan and coalescing represent just a few heuristic-based allocation approaches.

```
loadAI  r_arp, @a ⇒ r_a
loadI   2         ⇒ r_2
loadAI  r_arp, @b ⇒ r_b
loadAI  r_arp, @c ⇒ r_c
loadAI  r_arp, @d ⇒ r_d
mult    r_a, r_2  ⇒ r_a
mult    r_a, r_b  ⇒ r_a
mult    r_a, r_c  ⇒ r_a
mult    r_a, r_d  ⇒ r_a
storeAI r_a ⇒ r_arp, @a
```

(a) $a \leftarrow a * 2 * b * c * d$ in ILOC[16].

```
loadAI   r_arp,@a  ⇒ LR7
loadI    2         ⇒ LR8
loadAI   r_arp,@b  ⇒ LR6
loadAI   r_arp,@c  ⇒ LR4
loadAI   r_arp,@d  ⇒ LR2
mult     LR7,LR8   ⇒ LR5
mult     LR5,LR6   ⇒ LR3
mult     LR3,LR4   ⇒ LR1
mult     LR1,LR2   ⇒ LR0
storeAI  LR0 ⇒ r_arp,@a
```

(b) Code renamed into LRs.



(c) Live Range Spans.

Figure 3.3.: Live Ranges in a Basic Block [Cooper and Torczon, 2022, Fig. 13.1].

As for the allocation strategies used in this thesis, both frameworks already provide a multitude of premade implementations ready for use. LLVM offers four different register allocators while Cranelift uses the highly optimized backtracking allocator `regalloc2`[17]. The next sections will go over the high-level differences of both frameworks regarding register representation, register mapping and live range construction while exploring possible performance gains for just-in-time use as part of the Game Boy Advance emulation implementation.

LLVM

The general idea of LLVM's approach to register allocation utilizes aliasing, classification, direct and indirect mapping with some transformations and optimizations which are performed during allocation; [Pereira et al., 2006] will provide most of the information presented in this section. At first, LLVM assigns each physical register with an integer number ranging from 1 to 1023. The specific numbering scheme depends on the architecture though looking at the statically generated files which hold this information reveals a lexicographic mapping, that is, a register name starting with the letter `A` will be denoted by the number 1, `B` with 2 and so on. Furthermore, registers can be marked as *aliased* if they share the same location, for example, in x86 architectures where `EAX`, `AX` and `AL` share the first 8 bits. Then, they are grouped into classes in which elements are interchangeable and functionally equivalent such as 8-bit only registers. Virtual registers have a particular class only they can be mapped to. Going back to the 8-bit only example, this means that some virtuals are reserved purely for 8-bit allocation to those physical registers. During code generation but before register allocation, operands of

[16]The IR used in [Cooper and Torczon, 2022]: "Intermediate Language for an Optimizing Compiler".

[17]`https://github.com/bytecodealliance/regalloc2`

an instruction will consist mainly of virtual registers though physical registers may also be used. LLVM calls these registers *pre-colored* and uses them for passing parameters of function calls and imposing constraints on the allocator as they must not be overwritten by virtual values while alive.

To further simplify register allocation, LLVM performs a few transformations just before invoking the allocator. The SSA Deconstruction Phase replaces ϕ-instructions with other native semantic-preserving instructions, traditionally with `copy`-instructions. Instruction folding, then, is another such transformation which removes unnecessary `copy`-instructions to reduce possible register pressure. Afterwards, the actual allocation may begin with one of the chosen allocators. They work as follows [Pereira et al., 2006, see *Built in register allocators*]:

- **Fast.** Default for debug builds, allocates on a basic block level which is ideal for a block-level JIT. It attempts to keep values in registers as much as possible, reducing spills.
- **Basic.** Assigns live ranges one at a time based on some heuristics. Often used as a base class for more complex allocation schemes, seldom used directly in production.
- **Greedy.** This is the default allocator and extends the **Basic** allocator by incorporating global live range splitting. Intents to minimize spill code.
- **PBQP.** Partitioned Boolean Quadratic Programming. It constructs a PBQP problem, that is, an abstract optimization problem consisting of multiple choices with interdependencies, and invokes a solver before mapping the solution back to register assignment.

Unfortunately, while the user may freely choose between these allocators via a command-line argument when invoking `llc` (LLVM static compiler), it seems there is no option to configure which allocator to invoke via the LLVM C API, and thus no performance analysis may be carried out. Similarly, it would be equally as difficult to fairly compare the LLVM allocators with the Cranelift allocator as by that point, too many other factors would be in play. However, based on the given just-in-time constraints and the basic-block-level nature of emulation, the **Fast** allocator may be simple enough to be fast in allocation time while not needing the additional complexity.

Cranelift

Cranelift uses a backtracking allocator that computes precise live ranges, performs *merging* according to certain heuristics into "bundles" and then runs a loop assigning locations to these bundles, sometimes splitting them to further divide the problem. Once all locations are assigned, move instructions are inserted to connect everything [Fallin et al., 2021].

Before allocation begins, `regalloc2` expects some form of program input consisting of instructions with references to virtual registers arranged as a control flow graph. Constraints are added to let the allocator reason about register choice and usage of space, especially when dealing with ISA-specific instructions that implicitly access certain physical registers such as the x86 instruction `mul rcx`: The multiplicand is implicitly defined within `rax` and the result is stored there as well — if it is 128 bits wide, the upper 64 bits are stored in `rdx`. From the allocator's point of view, expecting virtual registers and SSA form, this cannot be represented without constraint annotations. These annotations may look like Listing 3.11 and will instruct the allocator to insert moves to and from the appropriate physical registers. Other annotations include `reuse-input(0)` for modifying existing values.

```
;; Multiplicand is implicitly in rax.
mul rcx  ; multiply by rcx
;; 128-bit wide result is implicitly placed rdx (high 64 bits)
;; and rax (low 64 bits).

;; Put inputs in v0 and v1. Use results in v2 and v3.
mul v0 [use, fixed rax], v1 [use, any reg],
    v2 [def, fixed rax], v3 [def, fixed rdx]
```

Listing 3.11.: Constraint example for `mul rcx` [Fallin, 2022b].

After `regalloc2` has received its expected input, it needs to be processed into a valid allocation problem. Live ranges, or rather, *liveness* of the virtual registers have to be computed via a backwards liveless analysis [Aho et al., 2006, p. 608], meaning any use of the virtual register propagates itself backward and ends at the first definition of itself. Out of this set of live virtual registers, actual live ranges are constructed. The allocator then defines so-called "bundles" as allocation units in which live ranges that fulfill certain heuristic criteria are merged together. Ranges spanning SSA block parameters, move instructions and inputs/outputs annotated with `reuse-input` make up this heuristic as long as they do not overlap.

Finally, these bundles join a priority workqueue and are processed one at a time, assigned locations or split further. Once a bundle is pulled from the queue, the allocator searches for a free physical register space inside its allocation map — "a `BTree` for fast lookups, that indicates whether the register is free or occupied at any program point and the liverange that occupies it" [Fallin, 2022b] — and allocates the bundle to the register if possible. If it finds a register which already has a bundle assigned to it, the bundle with the lower "spill cost" wins, otherwise the bundle is split and the process repeats.

While there would be little to no value in comparing the performance of `regalloc2` with one of the four LLVM register allocation methods because of the previously stated reasons, Cranelift's allocator began as a port of Mozilla's "IonMonkey" backtracking register and features a similar workflow to LLVM's **Greedy**-method. Some benchmark results[18] are available between the old "IonMonkey" allocator and `regalloc2` which indicate a faster compilation time of up to 42% and an improved execution time of up to 28%. Naturally, this does not translate 1:1 to potential speedups when compared to LLVM, but shows that improvements have been made that may also be applicable to their allocation strategies.

[18] `https://github.com/bytecodealliance/wasmtime/issues/3942`

3.5. Type System

Both frameworks make use of an extensive type system with strong and static typing. LLVM's type system tries to be as abstract as possible over all the architectures it supports yet also provides hardware-specific types or types for certain extensions only. By providing this higher level of abstraction, the programmer is granted the freedom to specify arbitrary integer sizes or an arbitrary amount of vector lanes; it is then LLVM's task how to map this to real hardware. Cranelift, on the other hand, is closer to the types a common ISA register could actually hold. Integer types are limited to powers of two between 8 and 128, only single and double precision floats are supported, no explicit pointer types nor aggregate types exist and vector lane sizes are also limited to powers of twos up to 256.

Table 3.1 presents the differences in a more succinct fashion. Cranelift follows a rather modern naming convention, akin to Rust, shortening type names as much as possible while still being easy to grasp. LLVM's type names and general structure seems more C-like and explicit.

	LLVM	CRANELIFT
INTEGER	• `iN` with `N` $\in [1, 2^{23}]$, e.g. `i134`	• `i8`, `i16`, `i32`, `i64`, `i128`
FLOAT	• `half (f16)`, `bfloat (f16*)` • `float (f32)`, `double (f64)` • `fp128`, `x86_fp80`, `ppc_fp128`	• `f32, f64`
SIMD / VECTOR	• `<vscale x N x type>`, e.g. ◦ `<8 x float>` ◦ `<vscale x 4 x ptr>`	• `iBxN`, `f32xN`, `f64xN` with: ◦ `B` $\in$ [`i8..i64`] ◦ `N` $\in [2, 256]$
MISC.	• Arrays: `[#elems x type]` Structs: `type { <list> }` • `void`, `function`, `amx`, `mmx`	• `imm64`, `offset32`, `ieee32`, `ieee64` • `intcc`, `floatcc` for `icmp` & `fcmp` • `-Inf`, `Inf`, `-NaN`, `NaN`

Table 3.1.: A comparison between the LLVM and Cranelift type system.

In terms of emulation, these differences do not matter much. Neither the additional abstraction nor the idealized ISA types really come into play since most of the time, the GBA only reads and writes either 8-, 16-, or 32-bit values. As such, instructions will only ever need the `i8`, `i16` or `i32` types in both frameworks[19]; in terms of API syntax: `context.iXXtype()` for LLVM and `types::IXX` for Cranelift where `XX` $\in [16, 32, 64]$. The signedness depends on the instructions.

3.6. Optimization Passes

An optimizer analyses the just-emitted code and rewrites it into a more efficient form to improve the quality of the code. Optimizations are implemented as *passes* that traverse a portion

[19]Minor exception for LLVM: `i1` must be used for boolean values.

of some code, *passing* over it so to speak, to either collect information or transform the program. Cranelift and LLVM both implement a number of optimization passes with varying degrees of complexity. Since Cranelift's main purpose is fast compilation, its passes are simpler and, by default, less are applied overall. LLVM is a sizable, mature and generic compiler framework, therefore it features a substantial amount of passes which it applies based on the chosen optimization level.

These levels may be chosen either via command-line arguments or within code utilizing the respective API calls. LLVM/`inkwell` defines four optimization levels: `OptimizationLevel::{None, Less, Default, Aggressive}`, not including special settings for code size optimization, to be specified when creating an `ExecutionEngine` with `Module::create_jit_execution_engine(opt_level)` for example. Cranelift supports three optimization levels: `opt_level::{none, speed, speed_and_size}` which can be specified via the `SettingsBuilder` API using `builder.set("opt_level", "<setting>")`. It is also possible to manually enable or disable every single optimization pass for more fine-grained control. By only considering the intersection of all possible optimizations provided by LLVM and Cranelift, a common baseline can be established for a fair performance evaluation.

Unreachable Code Elimination. Unreachable code describes code segments within a program that can never be executed as there exists no control flow path to the code from the rest of the program. The `cranelift_codegen` optimizer eliminates unreachable code regardless of optimization level, deleting whole blocks which cannot be reached from the entry block. LLVM offers a similar pass called `UnreachableBlockElim` which by itself just performs a depth-first traversal of the CFG and deletes the unvisited nodes. However, on its own it is never used unless explicitely called as this CFG simplification is part of a more complex pass called `SimplifyCFG` which removes basic blocks without predecessors and basic blocks with just an unconditional branch as well as merging blocks with only one predecessor and successor. `SimplifyCFG` is automatically included in `-O1` (or `OptimizationLevel::Less`).

This pass seeks to reduce code size and instruction cache usage. Unless there was an error in implementation, this pass is unlikely to affect much of the code generated for this project. Internal compiler transformations may also temporarily cause some code to be unreachable, although this usually resolves itself quickly.

Dead Code Elimination. Dead code does not affect program results. It differs from unreachable code since it can still be reached by the control flow path and execute instructions, unnecessarily increasing execution time. Dead variables arising from dead stores and loads can be considered a subset of dead code. Section 3.2 and the Static Single Assignment Form subsection already demonstrated that detecting dead code in SSA form is rather trivial. The classical elimination algorithm is similar to mark-sweep garbage collectors in that two passes over the IR are performed. *Mark* discovers the set of useful operations, *Sweep* removes useless operations.

Cranelift and LLVM both include a DCE pass as long as optimizations are enabled, regardless of level. Whereas LLVM incorporates DCE as a separate, toggleable pass — subdivided into single pass dead instruction elimination and dead code elimination as a second pass over the newly dead instructions — Cranelift removes instructions whose results are unused as part of its *e-graph* pass [Fallin, 2022a] among other transformations.

Global Value Numbering. Global value numbering (GVN) [Click, 1995; Gargi, 2002] attempts to replace instructions that each compute the same value with just a single instruction. By assigning a value number to variables and expressions, ideally reusing the same number for equivalent expressions, a value-number mapping may be created. It finds these equivalencies by looking for algebraic identities or finding identical operations with the same inputs, partitioning the computed results into *congruence classes*. Congruent values are guaranteed to yield the same result for all possible executions of the instruction routine. GVN lends itself well to constant folding and general redundant instruction elimination, such as redundant loads.

For example, this code `w := 3; x := 3; y := x + 4; z := w + 4` would constitute the following mapping: $\{w \mapsto 1, x \mapsto 1, y \mapsto 2, z \mapsto 2\}$. The optimizer would then be able to transform the previous code into `w := 3; x := w; y := w + 4; z := y`. Further optimizations might also remove the redundant assignments to `x` and `z`. Listing 3.12 depicts possible load elimination and store-to-load forwarding targets as a consequence of GVN: In both examples, the instruction sequences would compute the same values even if replaced with just one instruction since (a) `%P` does not change inbetween `loads` and (b) it is statically provable that `4` was just `stored` in `%P` rendering the `load` redundant.

```
%x = load i32* %P
%y = load i32* %P   ; <- Redundant
```

(a) Redundant load.

```
store i32 4, i32* %P
%a = load i32* %P ; <- %a equals 4
```

(b) Store-to-load forwarding.

Listing 3.12.: Redundant load and store elimination [Lattner, 2009].

LLVM implements an explicit GVN pass based on [Gargi, 2002], eliminating fully and partially redundant instructions as well as redundant loads. Cranelift features both an explicit pass for removing redundant loads, called `replace_redundant_loads`, only used for replacing loads with known values in memory in addition to applying a more general GVN pass during its e-graph construction phase [Fallin, 2022a].

Miscellaneous. Thanks to Cranelift's middle-end optimization framework based on e-graphs, or equivalency graphs [Willsey et al., 2021], many other optimizations like loop-invariant code motion (LCIM), constant folding, alias analysis and other specific *rewrite rules* such as constant propagation, algebraic reduction and strength reduction exist within the capabilities of its optimizer. These are, however, either part of the middle-end optimizer or happen very early during code generation, making it difficult to individually assess their contribution towards compilation speed or execution time as they cannot be enabled or disabled one at a time.

LLVM, though, allows full control over every single optimization pass; its `PassManager` takes in a stringified array of options identical to how one would pass the chosen passes to LLVM's opt-tool as a command-line argument[20]. Consequently, LCIM, constant folding and the reductions (scalar evolution) are all available as invidual and selectable passes.

[20]`https://thedan64.github.io/inkwell/inkwell/module/struct.Module.html#method.run_passes`

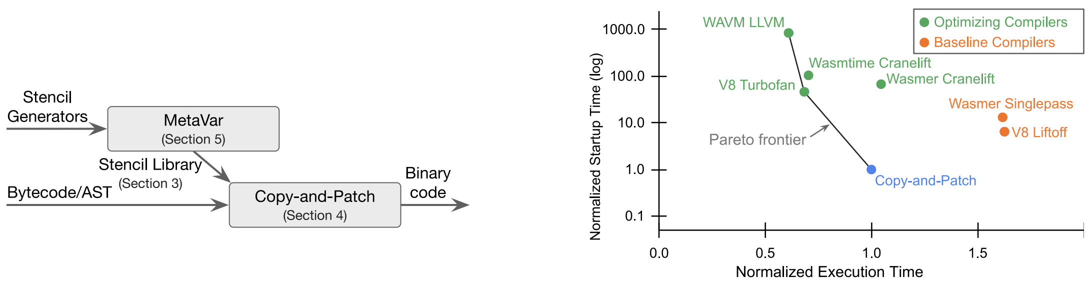
Chapter 4.

Related Work

In this chapter, related work in the area of just-in-time compilation, or more broadly dynamic recompilation, with a focus on emulation and performance analyses will be presented and discussed. In addition, their connection towards the work of this thesis will be evaluated.

Copy-and-Patch Compilation [Xu and Kjolstad, 2021]

Fast compilation is important, especially when the compilation occurs at runtime. Query compilers, virtual machines in web browsers and other real-time applications cannot afford the latency stemming from optimizing but slow compilers. Runtime execution environments typically contain several execution tiers, such as interpreters, baseline compilers and different optimization levels to balance startup delay and performance. [Xu and Kjolstad, 2021] propose a new algorithm for JIT-style baseline compilers called copy-and-patch. It uses so-called stencils to stitch together code from a library of binary implementation variants such that during code generation missing values must be inserted into the "holes" of these stencils.



(a) MetaVar architecture diagram. (b) PolyBench benchmarks.

Figure 4.1.: The MetaVar compiler and its copy-and-patch system and a scatter plot of normalized startup delay against execution time for seven WebAssembly compilers [Xu and Kjolstad, 2021, Fig. 4 & Fig. 2].

A pre-built library of composable and parametrizable binary code snippets provides these stencils. Optimization and code generation happen with a lookup table; the appropriate stencil is selected and instantiated by copying and patching in the missing values. Their contributions towards this approach are as follows:

1. The binary stencil concept — a pre-built implementation of an AST node or bytecode opcode with missing values like immediates, offsets, branch and call targets.

2. An algorithm which emits optimized machine code through the usage of a library with many binary stencil variants with two distinct types: one that enumerates different parameter configurations and one that enumerates different code patterns.
3. An algorithm for linearization of high-level language constructs like if-statements, generating machine code through composition of stencil fragments.
4. The MetaVar system (Figure 4.1a) for systematically generating binary stencils with C++ templates, leveraging Clang and LLVM to abstract away low-level platform-specific details.

The results seem to indicate significant improvements regarding both startup delay and execution performance. Xu and Kjolstad claim notable speedup compared with conventional baseline compilers and that it narrows the execution time gap to optimizing compilers, see Figure 4.1b. Code generation performance is consistently two to three orders of magnitudes faster compared to the other WASM compilers. [Xu and Kjolstad, 2021, Fig. 17 & Fig. 18] plot the absolute throughput in terms of megabytes of WASM code sections processed per second and the normalized log-scaled startup delay. On average, copy-and-patch processes code sections up to 8 times faster resulting in much lower startup delay. Execution performance is also drastically improved as, on average and depending on the benchmark, improvements of 40-60% have been measured in comparison to other baseline compilers. Optimizing compilers, such as Cranelift, still outperform copy-and-patch by a few percent, although the gap is closing.

An emulator might, therefore, make a good candidate for future copy-and-patch endeavours, given the much lower compilation time with potential room for subsequent optimizations. Creating and generating the appropriate stencils for the corresponding target, however, might lead to further complexity as they would have to accurately represent the guest ISA semantics. Already existing real-world applications include a state-of-the-art baseline Lua JIT compiler by [Xu, 2024] and `CPython` switching to a copy-and-patch compiler [Bucher, 2023].

A Dynamically Recompiling ARM Emulator [Sharp, 2001]

Attempting dynamic recompilation of an ARM processor as accurately as possible for use in emulation of larger computer systems, back in 2001, was not without its limitations and hurdles. Several commercial emulators existed but apart from brief white papers, they tended to limit their findings. Freely available emulators using ground-breaking techniques at the time remained closed-source and similarly tended not to fully release documentation. Open source emulators, on the other hand, while not lacking in source availability, used more primitive translation techniques unfit for full system emulation. [Sharp, 2001] sought to rectify this significant omission concerning computer system emulation using dynamic recompilation.

His TARMAC emulation project targets the ARM3 processor and generates x86 instructions. To aid conversion from the inherent complexity of ARM, an intermediate representation is used before emitting x86 code. Sharp calls this IR “armlets” of which one or more describe the accurate emulation of a single ARM instruction [Sharp, 2001, p. 40]. Armlets are a form of “three-address-code” [Aho et al., 2006, p. 363], that is, each armlet consists of at most one operator and three operands, normally one destination operand and two source operands. Given the expression $a + b * c - d$, a translated sequence of three-address-instructions might look as follows:

$$t_1 := b * c, \quad t_2 := a + t_1, \quad t_3 := t_2 - d \tag{1}$$

where t_1, t_2 and t_3 are compiler-generated temporary variable names. This mirrors the approach used in ARM instructions. Armlets, like their parent instructions, can therefore be written in the following format for simplicity:

```
@ Armlet Syntax:
<op> <dst>, <op1>, <op2>
@ Example & 3-address-form:
add x, y, z ; → x := y + z.
```

(a) Armlet syntax. [Sharp, 2001, p. 40]

```
@ [inflags->outflags] <armlet>
@ ex: adcs r1, r1, #0x1
[xxxx->xxxx] movc t0, 0x1
[xxCx->NZCV] adc r1, r1, t0
```

(b) Armlet example. [Sharp, 2001, p. 41]

Listing 4.1.: The textual form of armlets. Each armlet includes *inflags* and *outflags* to denote the condition flags it uses in the operation and which condition flags it will update. This assists target code generation.

Furthermore, Listing 4.1b shows that armlets operate on a set of variables. In order to represent the emulated state of an ARM processor, TARMAC chooses to expose the fifteen general purpose registers, the program counter, all four condition codes, the IRQ-Disable flag, the FIQ-Disable flag, the current mode and, additionally, thirty temporaries `t0` to `t29` for use as variable operands. As such, immediates must be loaded into a variable before being used. In total, there are five classes of armlets with different arguments, of which not all follow a three-address-structure [Sharp, 2001, p. 42]:

- **Implied:** No operands, e.g. `settrans`.
- **Transfer:** A variable and a 32-bit immediate, e.g. `movc r0, #0x12345678`.
- **2-Variable:** Two operand variables, e.g. `cmp r0, r1`.
- **3-Variable:** Three operand variables, e.g. `ror r0, r1, r2`.
- **Immediate:** One 32-bit immediate operand, e.g. `goto #0x87654321`.

Generating armlet sequences is done via what Sharp calls a “Profiler”; it takes a sequence of ARM instructions and generates the appropriate armlets before passing that on to the optimiser. As discussed in Section 2.2.3, recompiling only one instruction at a time would lead to unnecessary overhead which is why compilers typically operate on translation units such as basic blocks, functions or modules. Basic blocks lend themselves to easier optimizations but were considered as too “short” and “[it] would also mean returning to the dispatcher at the end of every iteration of a short loop” [Sharp, 2001, p. 43]. Instead, TARMAC uses its own translation unit, aptly named “chunk”, to describe a sequence of instructions. A chunk, like a basic block, has only one entry point but possibly multiple exit points. Instructions inside a chunk are classified into four categories based on their effect on control flow [Sharp, 2001, p. 43]: *unaffecting*, *branch-inside-chunk*, *branch-outside-chunk*, *pc-adjusting*. Unaffecting instructions are simply added to the chunk, branch-inside-chunk instructions are handled by generating a goto to an armlet within the chunk. Branch-outside-chunk and PC-adjusting instructions drop back to the dispatcher to decide whether to continue with the current chunk.

This decision process is outlined in Figure 4.2. Only if the dispatcher decides not to continue the chunk and the instruction is not conditionally executed does the chunk end. Thus, unlike basic blocks which always end on a branch whether it is taken or not, a chunk only ends on a satisfied conditional branch-outside-chunk or an unconditional branch-outside-chunk. The main advantages of using this less constrained translation unit are longer sequences of code and therefore less recompilation overhead. On the other hand, detecting a valid chunk requires more work than detecting a valid basic block.

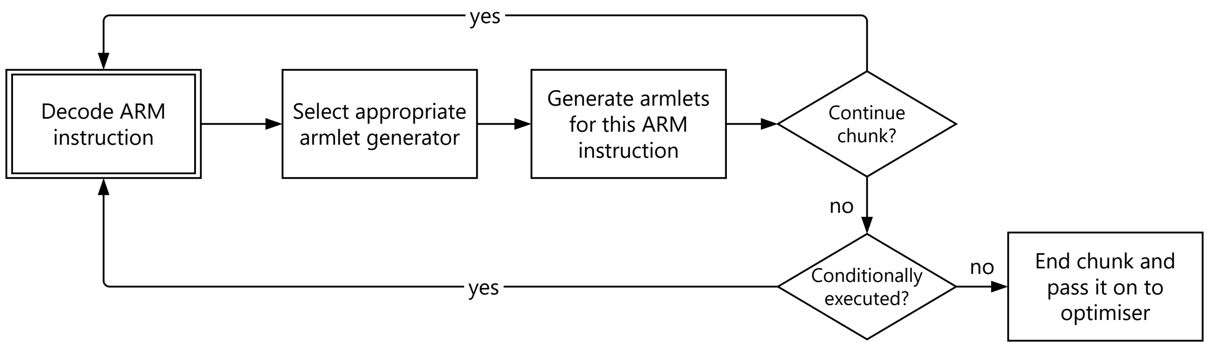

yes
Decode ARM instruction
Select appropriate armlet generator
Generate armlets for this ARM instruction
Continue chunk?
no
Conditionally executed?
no
End chunk and pass it on to optimiser
yes

Figure 4.2.: Outline of the decision process concerning whether to end the current chunk, based on [Sharp, 2001, Fig. 20].

Sharp foregoes many common optimization techniques on the basis that "they are likely to have already been performed on the ARM code and so would have little benefit" [Sharp, 2001, p. 54]. Nonetheless, he employs some techniques that capitalize on the design philosophy of armlets. Conditional blocks, for example, are often realized through several consecutive conditionally executed ARM instructions which does not translate well to x86 as its support for conditional execution is not as comprehensive. A naive recompiler would then insert conditional branches before each conditional instruction which is unnecessary as the result of all condition checks would be equal. In order to mitigate this issue, the TARMAC profiler inserts a goto with the inverse condition before the armlets, backpatching its location once known, resulting in less branches overall. Another useful optimization is redundant condition flag calculation elimination — removing the calculation of flags after every instruction if the result is never used.

Ultimately, as a result of this novel approach to ARM dynamic recompilation, single RISC ARM instructions can sometimes be translated to a single CISC x86 instruction. The aforementioned optimizations, including those done during code generation like constant folding and propagation, are successful in eliminating other ARM obscurities and the profiler is able to translate ARM to armlet in a single-pass visit [Sharp, 2001, pp. 72–73]. However, no performance analysis seems to have occured as no measurements have been provided in Sharp's report. Nonetheless, some variation of this design approach could lend itself well to newer emulation endeavours as expanding the concept of traditional IRs to include additional information and slightly different constraints in addition to the use of "chunks" might open up new avenues for code analysis, optimization or IR design.

Emulator Speed-up Using JIT and LLVM [Wennborg, 2010]

Utilizing a preexisting compiler framework with readily available optimization passes, such as LLVM, for building a JIT compiler applied to a video-coding system emulator could lead to rather notable execution time improvements if implemented correctly. [Wennborg, 2010] investigates this possibility in his thesis by evaluating existing passes, patching others and implementing new ones as well. The focus of his work is on improving total execution time, or net speed-up, with less concern for extra startup delay as long as overall execution time is reduced. Wennborg's system of choice is a video engine developed by ARM Sweden, responsible for encoding and decoding video data in products such as phone cameras which had used emulation through simple interpretation of firmware code to aid development [Wennborg, 2010, p. 3].

Specifically, the thesis is concerned with the emulation of the RASC[21] processor as part of the larger emulation system. For JIT compilation, the RASC instruction sequences are translated to LLVM instructions at a function-level. Instructions within these functions operate on a state object which is passed in when calling the function, modifying it accordingly. A separate MMU emulator handles memory access and translation between the RASC and host address space. According to [Wennborg, 2010, pp. 11–12], function objects should generally match firmware behaviour, returning zero if all went well. However, certain situations like indirect jumps with destinations only determined at runtime or an explicit call to suspend emulation via the `wait` instruction cause the LLVM function objects to no longer match with firmware behaviour. The function builder, then, either elects to compile the LLVM IR into native machine code for subsequent execution, as shown in Figure 4.3, or returns the special value of one to signal a "get-out-of-emulation return" [Wennborg, 2010, p. 12].

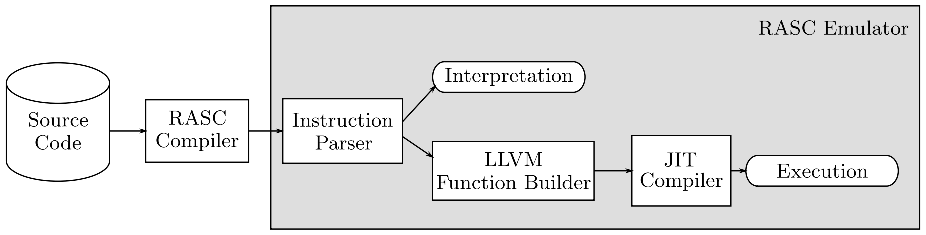


Figure 4.3.: Flow from RASC source code to execution [Wennborg, 2010, Fig. 4.2].

Removing Redundant Memory Access. One of the first optimization opportunities Wennborg looked at was the reduction of memory access instructions with no subsequent effects on the generated code. This, typically, includes dead load elimination as well as dead store elimination though in order to properly detect these cases, a process called "Alias Analysis" happens first. Alias analysis is responsible for figuring out whether two pointers may point to the same object in memory or not. LLVM, by default, implements a basic form of this process but omits two cases which commonly occur in the RASC JIT compiler: *null pointer elimination* and *constant pointer elimination*. LLVM is unable to detect if a null pointer and and a real pointer could alias and thus fails to remove dead loads in cases where a second load

[21]Developed specifically for the video engine; presumably a modified ARM microcontroller.

would be redundant after a function call. In a similar fashion, LLVM cannot properly detect aliasing if it involves constant addresses [Wennborg, 2010, pp. 17–18].

These missing heuristics were retrospectively added to the alias analysis process as a result of Wennborg implementing patches for both, leading to more aggressive detection of dead stores and dead loads which can then be eliminated. All further measurements in his thesis were conducted with these patches applied.

Reducing MMU Emulation. As loads and stores are handled by a separate MMU emulator, each generated function call has to decide if the address represents a hardware register which activates yet another separate hardware emulator or if it has to translate the address space inside the memory mapping. This leads to "considerable" overhead and it would be "desirable to replace as many of these instructions as possible with native loads and stores" [Wennborg, 2010, p. 23]. Some approaches to alleviate this overhead are: *compile-time constant address translation*, *moving address translation out of loops* and *combining translation of close addresses*.

Compile-Time Constant Address Translation detects if an address in a call to the MMU emulator is constant with the intent of doing address translation at compile-time; statically allocated data, such as fixed-size arrays, fulfills this criteria. The other two optimizations mostly work to reduce the absolute number of MMU calls necessary, either by removing calls out of loops or combining calls when the addresses refer to the same structure.

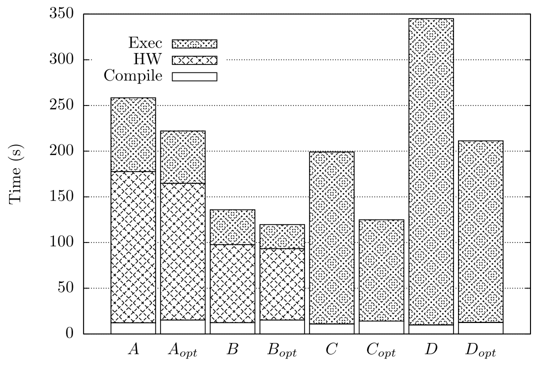
Loop-Invariant Code Motion. Scalar promotion, hoisting or loop-invariant code motion (LICM) is a transformation that takes an expression inside a loop body which yields the same result regardless of the amount of iterations and evaluates it once before the loop [Aho et al., 2006, p. 592]. LLVM already implements LICM — its effects on the RASC JIT depend on the benchmark (Figure 4.4), mostly benefitting the β decoder [Wennborg, 2010, Table 5.9].

Function Inlining. Function inlining replaces function calls with the bodies of the called functions. This removes call overhead and exposes more code to the optimizer at the cost of increased code size. More code means more compile-time and more pressure on the instruction cache. While LLVM does provide a built-in pass for inlining, according to [Wennborg, 2010, p. 35], it only works on entire modules and is therefore unsuitable for the emulator's gradual approach in building modules. Wennborg implements a custom inlining pass with minor speed-ups.

Figure 4.4 showcases the final results of all optimizations — the previous sections illustrated only the most prominent ones — with varying improvements depending on the benchmarking format. More significant speedups were observed with the β decoder throughout, yet overall every benchmark was able to reduce its execution time while keeping compile-time low. What this shows is, that careful consideration with respect to optimization choice may lead to satisfactory results even when the problem statement, that is, improving emulation and JIT compilation speed, requires a light-weight solution. Additionally, Wennborg's thesis demonstrates the importance of benchmarking every atomic change to code generation as to evaluate whether certain optimizations provided a measurable speedup and, if they did, which permutation leads to the best results.

Benchmark	Format	Resolution	Frames
A	α	720×480	455
B	α	720×526	219
C	β	720×480	455
D	β	1280×720	320

(a) Benchmark Data.
[Wennborg, 2010, Table 3.1]



(b) Optimization Results.
[Wennborg, 2010, Fig. 6.1]

Figure 4.4.: The four different benchmarks and their execution time before and after applying optimizations. It consists of decoding video files in different formats. The α decoder is hardware accelerated while the β decoder is implemented in software.

Chapter 5.

Evaluation

In this chapter, a few potential metrics for comparing the code generation and performance of the Cranelift and LLVM JIT implementations will be introduced. Additionally, a short-form evaluation will be carried out and the results used to extrapolate some conclusions.

5.1. Setup

5.1.1. Idea and Tooling

For this evaluation, the raw disassembly of the generated x86 code and uncapped performance were used as the main criteria for determining a framework's suitability towards emulation. Compile time by itself, and potential subsequent stutter, could not be accurately measured as the benchmarking scenarios were too limited in scope, however, looking at the average of the uncapped performance can still reveal extreme outliers if only at a cursory glance.

To access the disassembled code, Cranelift offers either a dedicated `compile_and_emit` method which takes in a mutable byte buffer in which it emits the final machine code in byte form or, more importantly, a human readable form inside an `Option<String>` struct member inside a `CompiledCode` struct — `Context::set_disasm()` has to be set to `true`, otherwise the option type returns `None`. LLVM/`inkwell` requires the manual creation of the `TargetMachine` struct, which then offers the `write_to_{file, memory_buffer}` methods. Simple *diffing* is then used to spot visible differences in the output as more and more optimizations are gradually enabled.

Performance is measured in frames per second with the emulation loop being executed as fast as possible. A ring buffer keeps track of the timestamps at which each frame first finished to render, producing a rough estimate of the average frame time.

5.1.2. Benchmark ROMs / Scenarios

Testing the aforementioned criteria is done via a simple demo ROM meant to demonstrate basic video output with as few instructions as possible. The ROM, commonly referred to as `gang.gba` or `first-1.gba` [Peach, 2024], only consists of 25 instructions in total, six unique opcodes and covers three instruction families, see Listing 5.1. When run, it outputs a stripe pattern over the whole screen with slight variations in color, as can be seen in Figure 5.1.

```
mov  r0, #4          @ 0x0000000000000000:  04 00 A0 E3
lsl  r0, r0, #0x18   @ 0x0000000000000004:  00 0C A0 E1
mov  r1, #0x400      @ 0x0000000000000008:  01 1B A0 E3
add  r1, r1, #3      @ 0x000000000000000c:  03 10 81 E2
strh r1, [r0]        @ 0x0000000000000010:  B0 10 C0 E1
mov  r1, #0x7c00     @ 0x0000000000000014:  1F 1B A0 E3
mov  r2, #0x3e0      @ 0x0000000000000018:  3E 2E A0 E3
mov  r3, #0x1f       @ 0x000000000000001c:  1F 30 A0 E3
mov  r0, #6          @ 0x0000000000000020:  06 00 A0 E3
lsl  r0, r0, #0x18   @ 0x0000000000000024:  00 0C A0 E1
mov  r4, #0x300      @ 0x0000000000000028:  03 4C A0 E3
lsl  r4, r4, #7      @ 0x000000000000002c:  84 43 A0 E1
add  r4, r4, r0      @ 0x0000000000000030:  00 40 84 E0
strh r1, [r0]        @ 0x0000000000000034:  B0 10 C0 E1
add  r0, r0, #2      @ 0x0000000000000038:  02 00 80 E2
add  r1, r1, #0x80   @ 0x000000000000003c:  80 10 81 E2
strh r2, [r0]        @ 0x0000000000000040:  B0 20 C0 E1
add  r0, r0, #2      @ 0x0000000000000044:  02 00 80 E2
add  r2, r2, #0x80   @ 0x0000000000000048:  80 20 82 E2
strh r3, [r0]        @ 0x000000000000004c:  B0 30 C0 E1
add  r0, r0, #2      @ 0x0000000000000050:  02 00 80 E2
add  r3, r3, #0x80   @ 0x0000000000000054:  80 30 83 E2
cmp  r0, r4          @ 0x0000000000000058:  04 00 50 E1
blt  #0x34           @ 0x000000000000005c:  F4 FF FF BA
b    #0x60           @ 0x0000000000000060:  FE FF FF EA
```

Listing 5.1.: Disassembly of `gang.gba`. Only three different instruction families and six unique opcodes are used (Data Processing, Half-Word Store/Load, Branch).

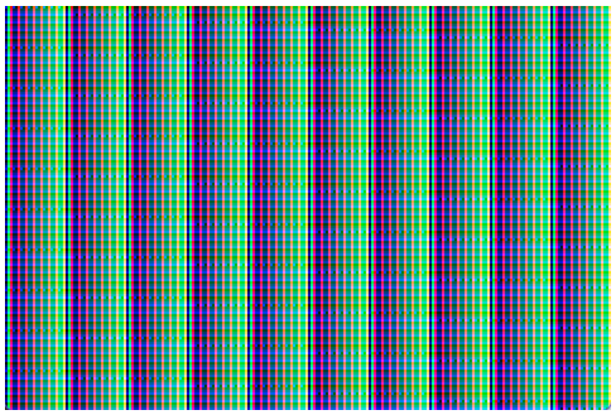


Figure 5.1.: Visual output of `gang.gba`. The main loop produces a repeating stripe pattern.

5.2. Results

5.2.1. Code Generation with Cranelift

One of the very first instructions inside a Game Boy Advance ROM, once the BIOS has concluded, is a branch jump 192 bytes forward, skipping over the cartridge header to the actual ROM start. This already constitutes a basic block as the sequence ends with an exit branch, that is, itself. As such, the Cranelift JIT stops collecting ARM instructions and begins to translate into native machine code. The Cranelift IR for this branch forward, as well as two slightly different machine code translations can be seen in Listing 5.2. Block 0 carries one block parameter `v0` which represents a mutable pointer to an array of all guest general purpose registers, such that any register can be accessed via a byte offset `load`/`store` from `v0`. Line 3 in Listing 5.2a shows this design choice: `v1` gets assigned the value at address `v0+60`, that being PC or `r15` as each register is 32 bits or 4 bytes in size and $60/4 = 15$. Lines 4–6, then, implement the actual opcode behaviour, adding a constant offset to PC and aligning it to a four byte boundary.

```
1 function u0:0(i64) -> i64 windows_fastcall {
2 block0(v0: i64):
3     v1 = load.i32 v0+60
4     v2 = iadd_imm v1, 8
5     v3 = iadd_imm v2, 184
6     v4 = band_imm v3, -4
7     store v4, v0+60
8     return v0
9 }
```

(a) Cranelift IR.

```
block0:
  movl    60(%rcx), %r8d
  movl    $184, %r9d
  lea     8(%r8,%r9,1), %edx
  andl    %edx, const(0), %edx
  movl    %edx, 60(%rcx)
  movq    %rcx, %rax
  movq    %rbp, %rsp
  popq    %rbp
  ret
```

(b) x86 with `opt_level=none`.

```
block0:
  ;; Constant Folding.
  movl    $192, %eax
  addl    %eax, 60(%rcx), %eax
  andl    %eax, const(0), %eax
  movl    %eax, 60(%rcx)
  movq    %rcx, %rax
  movq    %rbp, %rsp
  popq    %rbp
  ret
```

(c) x86 with `opt_level=speed`.

Listing 5.2.: The very first instruction of `gang.gba`, or most GBA software, compiled to Cranelift IR, x86 without optimizations and x86 with slight optimizations.

Listing 5.2b and Listing 5.2c display the disassembled x86 code. Besides the necessary alignment of the stack pointer, both examples show a close 1:1 translation to their input. As this code was generated on a Windows machine, the Windows FastCall calling convention was used in which the `rcx` register represents the first argument of a function (`v0`) and `rax` its return value. A similar offset behaviour can be observed. However, such a near identical translation is seldom ideal, hence, optimizations were enabled for Listing 5.2c: It is immediately observable that a *constant folding* pass was applied since the aforementioned offset of 192 bytes no longer has to be calculated during runtime. Essentially, the code boils down to loading the value 192 into `eax`, adding the value 60 bytes offset from `rcx` onto `eax` and storing the result in `rcx`.

The ROM features two additional basic blocks which can be seen in Listing 5.1: Lines 1–24 which make up the whole program and line 25, providing an endless loop. This project manages to compile and emit code for the rest of the ROM, turning 25 lines of ARM assembly into roughly 150 lines of x86 assembly, however at its current state, missing implementation details keep it from showing any visual output. Nevertheless, tracking the registers and their change over time reveals a mostly working implementation when compared with debugging tools and the source code itself.

5.2.2. Code Generation with LLVM

No working LLVM implementation currently exists due to time constraints. Even so, as showcased in Listing 3.7, the basic idea and syntax of the API does not differ all that much from the existing Cranelift code base. Based on the research presented in prior chapters, it is reasonable to assume that LLVM would produce much more optimized code — even at low optimization levels — at the cost of compile time.

Chapter 6. Conclusion

This chaper reiterates the main talking points of this thesis, what was achieved, and discusses potential future work introducing novel concepts that were out of scope for this work.

6.1. Summary

As hardware complexity increases over time and CPUs become faster and more complex to emulate, common emulation implementation techniques no longer manage to reach the native performance of the guest device. Modern processor emulators, therefore, employ a technique called *just-in-time compilation* to keep up with the system's demands. This enables the possibility of near-native performance as common optimization passes can be applied, decreasing execution time but increasing compilation time. However, this trade-off has to be considered carefully as too steep of an increase in compilation time could offset the performance gained from executing native code.

This thesis seeks to evaluate this balance and possibly find a conclusive answer by providing an introduction not only into the world of emulation and its implementation techniques, but also into the hardware internals of the Game Boy Advance which represents the emulated device used for this evaluation. Two CPU implementations were developed — using LLVM and Cranelift — and assessed for their compile speed and code quality to find an adequate trade-off between these metrics. By utilizing the respective APIs of both frameworks, generating their intermediate representation for each guest instruction and individually selecting certain optimization passes to run, a common baseline for comparison was established. The rest of the system was emulated as well, however, it played no significant part in this evaluation.

To demonstrate the concrete differences in the code generation of each framework, each JIT prints the raw disassembly after a block was translated and the uncapped performance, in frames per second, was tracked. Whenever possible, optimizations were gradually enabled and explored.

6.2. Future Work

Scheduler. As alluded to in Section 3.1, tick-based component cycling wastes a lot of performance by checking many conditions per component each cycle which, more often than not, do not satisfy all criteria to perform their update routine. Interrupts, for example, are typically checked before each instruction even if some of them may only be dispatched during certain hardware events. Another example comes from the PPU: As it cycles through different modes, internal registers receive mode-specific updates, IRQ conditions are evaluated and scanlines are drawn. With the current design, this cycling is implemented as a state machine that ticks every cycle, checks the current mode and whether a certain cycle count is reached to run some code, subsequently switching modes.

An event-based scheduler system would allow for precise management of future hardware events by enqueueing function callbacks in a priority queue with corresponding timestamps. A global cycle counter is updated and used as a key, while only the CPU is ticked until the timestamp of the closest event is hit; advancing directly to it. This event is then dequeued and its callback executed. Essentially, the scheduler is responsible for verifying whether it is time to update components and not the components themselves, alleviating the need for constant cycle checks.

Full JIT. This thesis only implements a minimal subset of the ARMv4T instruction set[22] needed to run the most simple of test ROMs as to provide a proof of concept for evaluation. Although enough differences can be made out to reach a verdict regarding code generation even with this minimal implementation, it would nonetheless elevate this work had both JITs been developed to completion. Complex pseudo-3D homebrew demo ROMs could have been used for a more thorough performance analysis, since the calculations necessary to achieve those effects would showcase an ideal CPU workload. Unfortunately, time constraints and issues during implementation led to a rather limited proof-of-concept.

Idle Loop Detection. Many games use idle loops, or wait loops, to wait for certain hardware events to happen before executing certain routines. Waiting for V-Blank, for instance, is a common way to determine if the drawing phase of a frame has ended such that new calculations may be performed. Likewise, some routines may use idle loops to repeat an action as long as a certain value is different than its expected value. This uses up unnecessary processor time, especially on the GBA which has dedicated facilities to reduce power consumption while waiting for hardware events — however, not all games make use of this best practise.

Idle loop detection is an emulation-specific optimization designed to detect and, if possible, skip these loops if it can determine their final state. Usually, the detection of idle loops consists of a trivial backwards branch jump check, that is, everytime a conditional branch jumps backwards, keep track of the new PC and if another one happens, evaluate `new_pc == tracked_pc` which yields true if a loop was found. Detecting the inner content of the loop and its final state is more difficult, relying on ISA-specific heuristics and edge cases to be aware of.

[22]Specifically, only the JITs are incomplete. The baseline interpreter implements 100% of the ISA.

Bibliography

[Advanced Risc Machines Ltd., 1995] Advanced Risc Machines Ltd. (1995). *ARM7TDMI Data Sheet* (No. ARM DDI 0029E). Retrieved from `https://www.dwedit.org/files/ARM7TDMI.pdf` (Cited on pages 3, 4, 5, 6, 7, 8, 65 and 66)

[Aho et al., 2006] Aho, A. V., Lam, Monica S., Sethi, R., and Ullman, J. D. (2006). *Compilers - Principles, Techniques & Tools, 2nd Edition.* Addison-Wesley. (Cited on pages 39, 44 and 48)

[Analog Devices Inc., 2005] Analog Devices Inc. (2005, October). Precision Analog Microcontroller, 12-Bit Analog I/O, ARM7TDMI MCU, page 40. (Cited on page 4)

[ARM Limited, 2001] ARM Limited. (2001, April 17). ARM7TDMI Technical Reference Manual. (Cited on pages 3 and 4)

[Arpaci-Dusseau and Arpaci-Dusseau, 2023] Arpaci-Dusseau, R. H., and Arpaci-Dusseau, A. C. (2023). *Operating Systems: Three Easy Pieces*, page 213. Arpaci-Dusseau Books. (Cited on page 17)

[Aycock, 2003] Aycock, J. (2003). A brief history of just-in-time. In *ACM Comput. Surv.*,Vol. 35, page 97. New York, NY, USA: Association for Computing Machinery. (Cited on page 19)

[bjorn3 and Singh, 2020] bjorn3, and Singh, S. M. (2020, March 18). Cranelift IR Reference. Retrieved from `https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/docs/ir.md` (accessed May 10, 2024) (Cited on pages 34 and 68)

[Braun and Hack, 2009] Braun, M., and Hack, S. (2009). Register Spilling and Live-Range Splitting for SSA-Form Programs. In *Compiler Construction*, pages 174–189. Berlin, Heidelberg: Springer Berlin Heidelberg. (Cited on page 37)

[Bucher, 2023] Bucher, B. (2023, December 25). GH-113464: A copy-and-patch JIT compiler. Retrieved from `https://github.com/python/cpython/pull/113465` (accessed May 14, 2024) (Cited on page 44)

[Bytecode Alliance, 2024] Bytecode Alliance. (2024). Cranelift - A Bytecode Alliance project. Retrieved from `https://cranelift.dev/` (accessed March 18, 2024) (Cited on page 21)

[Cabrera-Arteaga et al., 2024] Cabrera-Arteaga, J., Fitzgerald, N., Monperrus, M., and Baudry, B. (2024). Wasm-Mutate: Fast and effective binary diversification for WebAssembly. *Computers & Security*, *139*, 103731. (Cited on page 21)

[Clark and Schneidereit, 2020] Clark, L., and Schneidereit, T. (2020, October 26). Improving testing with fuzzing. Retrieved from `https://bytecodealliance.org/articles/1-year-update#improving-testing-with-fuzzing` (accessed March 18, 2024) (Cited on page 21)

[Click, 1995] Click, C. (1995). Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 246–257. La Jolla, California, USA: Association for Computing Machinery. (Cited on page 42)

[Cooper and Torczon, 2022] Cooper, K. D., and Torczon, L. (2022). *Engineering a Compiler, Third Edition.* Morgan Kaufmann Publishers Inc. (Cited on pages 20, 30, 36 and 37)

[Copetti, 2019] Copetti, R. (2019, August 18). Game Boy Advance Architecture - A Practical Analysis. Retrieved from `https://www.copetti.org/writings/consoles/game-boy-advance/` (accessed January 5, 2024) (Cited on pages 3 and 9)

[Cytron et al., 1991] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, *13*(4), 451. (Cited on page 29)

[de Moraes, 2022] de Moraes, R. B. (2022, November 26). Compiling Brainfuck code - Part 3: A Cranelift JIT Compiler. Retrieved from `https://rodrigodd.github.io/2022/11/26/bf_compiler-part3.html` (accessed May 2, 2024) (Cited on page 28)

[Deutsch and Schiffman, 1984] Deutsch, L. P., and Schiffman, A. M. (1984). Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 297–302. Association for Computing Machinery. (Cited on page 19)

[Fallin, 2021] Fallin, C. (2021, March 15). Cranelift, Part 3: Correctness in Register Allocation. Retrieved from `https://cfallin.org/blog/2021/03/15/cranelift-isel-3/` (accessed March 18, 2024) (Cited on pages 21 and 36)

[Fallin, 2022b] Fallin, C. (2022b, June 9). Cranelift, Part 4: A New Register Allocator. Retrieved from `https://cfallin.org/blog/2022/06/09/cranelift-regalloc2/` (accessed May 13, 2024) (Cited on page 39)

[Fallin, 2022a] Fallin, C. (2022a, June 30). Cranelift: Using E-Graphs for Verified, Cooperating Middle-End Optimizations. Retrieved from `https://github.com/bytecodealliance/rfcs/pull/27` (accessed March 18, 2024) (Cited on pages 21, 41 and 42)

[Fallin et al., 2021] Fallin, C., Elliott, T., and d'Antras, A. (2021, June 18). regalloc2 Design Overview. Retrieved from `https://github.com/bytecodealliance/regalloc2/blob/main/doc/DESIGN.md` (accessed April 11, 2024) (Cited on page 38)

[Gargi, 2002] Gargi, K. (2002). A sparse algorithm for predicated global value numbering. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 45–56. Berlin, Germany: Association for Computing Machinery. (Cited on page 42)

[Gohman, 2018] Gohman, D. (2018, November 28). Cranelift compared to LLVM. Retrieved from `https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/docs/compare-llvm.md` (accessed March 18, 2024) (Cited on pages 21 and 30)

[Granett, 2000] Granett, D. (2000, August 24). Game Boy Advance: It's Finally Unveiled. Retrieved from `https://www.ign.com/articles/2000/08/24/game-boy-advance-its-finally-unveiled` (accessed January 15, 2024) (Cited on pages 3 and 8)

[Hayes, 1978] Hayes, J. P. (1978). *Computer Architecture and Organization*, pages 505–506. McGraw-Hill, Inc., edition 3. (Cited on page 8)

[Horspool and Marovac, 1978] Horspool, R. N., and Marovac, N. (1978, October). An approach to the problem of detranslation of computer programs. McGill University. (Cited on page 20)

[Jack and Tsatsoulin, 2002] Jack, K., and Tsatsoulin, V. (2002). *Dictionary of Video and Television Technology*. Elsevier Science. (Cited on page 11)

[Klabnik and Nichols, 2022] Klabnik, S., and Nichols, C. (2022). *The Rust Programming Language, 2nd Edition*, page 177. No Starch. (Cited on page 18)

[Kolsoi, 2023] Kolsoi, D. (2023). inkwell: It's a New Kind of Wrapper for Exposing LLVM (Safely). Retrieved from `https://github.com/TheDan64/inkwell` (accessed May 7, 2024) (Cited on page 31)

[Koninklijke Bibliotheek, 2006] Koninklijke Bibliotheek. (2006, October). What is emulation?. Retrieved from `https://web.archive.org/web/20151024005119/https://www.kb.nl/en/organisation/research-expertise/research-on-digitisation-and-digital-preservation/emulation/what-is-emulation` (Cited on page 16)

[Korth, 2001] Korth, M. (2001). GBATEK - Gameboy Advance / Nintendo DS / DSi / 3DS - Technical Info. Retrieved from `https://problemkaputt.de/gbatek.htm` (accessed February 26, 2024) (Cited on pages 13 and 15)

[Kulot, 2023] Kulot, K. (2023, December 23). Bus: Paging for faster jump table. Retrieved from `https://github.com/xkevio/kba/commit/cd29253a5153e642e0c221d180d59234e2377c96` (Cited on page 18)

[Lantinga, 1997] Lantinga, S. (1997). SDL Wiki: SDL_TextureAccess. Retrieved from `https://wiki.libsdl.org/SDL2/SDL_TextureAccess` (accessed April 29, 2024) (Cited on page 24)

[Lattner, 2002] Lattner, C. (2002, December). *LLVM: An Infrastructure for Multi-Stage Optimization*. Master's thesis. Retrieved from `https://llvm.org/pubs/2002-12-LattnerMSThesis.pdf` (Cited on page 20)

[Lattner, 2009] Lattner, C. (2009, December 17). Introduction to load elimination in the GVN pass. Retrieved from `https://blog.llvm.org/2009/12/introduction-to-load-elimination-in-gvn.html` (accessed May 14, 2024) (Cited on page 42)

[Lattner and Adve, 2004] Lattner, C., and Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. (Cited on page 21)

[Lattner and Adve, 2012] Lattner, C., and Adve, V. (2012). LLVM Language Reference Manual. Retrieved from `https://llvm.org/docs/LangRef.html#introduction` (accessed May 6, 2024) (Cited on pages 30 and 31)

[Next Generation, 1996] Next Generation. (1996, March). The Next Generation 1996 Lexicon A to Z: Mode 7, page 37. Imagine Media. (Cited on page 11)

[Nystrom, 2021] Nystrom, R. (2021). *Crafting Interpreters*, page 18. Genever Benning. (Cited on page 16)

[Osborne, 1980] Osborne, A. (1980). *An Introduction to Microcomputers - Volume 1.* Osborne/McGraw-Hill, edition 2. (Cited on pages 14 and 15)

[Peach, 2024] Peach. (2024). Panda3DS: HLE 3DS emulator (CDN branch). Retrieved from `https://github.com/wheremyfoodat/Panda3DS/tree/cdn/docs/gba-demos` (accessed May 17, 2024) (Cited on page 51)

[Pereira et al., 2006] Pereira, F. M. Q., Wendling, B., and Eveson, S. (2006, September). The LLVM Target-Independent Code Generator: Register Allocation. Retrieved from `https://www.llvm.org/docs/CodeGenerator.html#register-allocator` (accessed April 8, 2024) (Cited on pages 37 and 38)

[Pugh, 1995] Pugh, E. (1995). *Building IBM: Shaping an Industry and Its Technology.* Cambridge, Mass. (Cited on page 15)

[Rosen et al., 1988] Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1988). Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, page 12. San Diego, California, USA: Association for Computing Machinery. (Cited on page 29)

[Sharp, 2001] Sharp, D. (2001). *TARMAC: A Dynamically Recompiling ARM Emulator.* Retrieved from `https://www.davidsharp.com/tarmac/tarmacreport.pdf` (Cited on pages 44, 45 and 46)

[Thain, 2020] Thain, D. (2020). Chapter 8 – Intermediate Representation. In *Introduction to Compilers and Language Design, 2nd Edition*, pages 119–134. (Cited on pages 27 and 29)

[Tucker, 1965] Tucker, S. G. (1965). Emulation of large systems. *Commun. ACM*, *8*(12), 753. (Cited on pages 15 and 16)

[VanHattum et al., 2024] VanHattum, A., Pardeshi, M., Fallin, C., Sampson, A., and Brown, F. (2024). Lightweight, Modular Verification for WebAssembly-to-Native Instruction Selection. (Cited on page 21)

[Vijn, 2005] Vijn, J. (2005, June). TONC: GBA Extended. Retrieved from `https://www.coranac.com/tonc/text/toc.htm` (accessed February 2, 2024) (Cited on page 14)

[Wennborg, 2010] Wennborg, H. (2010, January). *Emulator Speed-up Using JIT and LLVM.* Master's thesis. Retrieved from `https://llvm.org/pubs/2010-01-Wennborg-Thesis.pdf` (Cited on pages 47, 48 and 49)

[Willsey et al., 2021] Willsey, M., Nandi, C., Wang, Y. R., Flatt, O., Tatlock, Z., and Panchekha, P. (2021). egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages*, *5*(POPL), 1–29. (Cited on page 42)

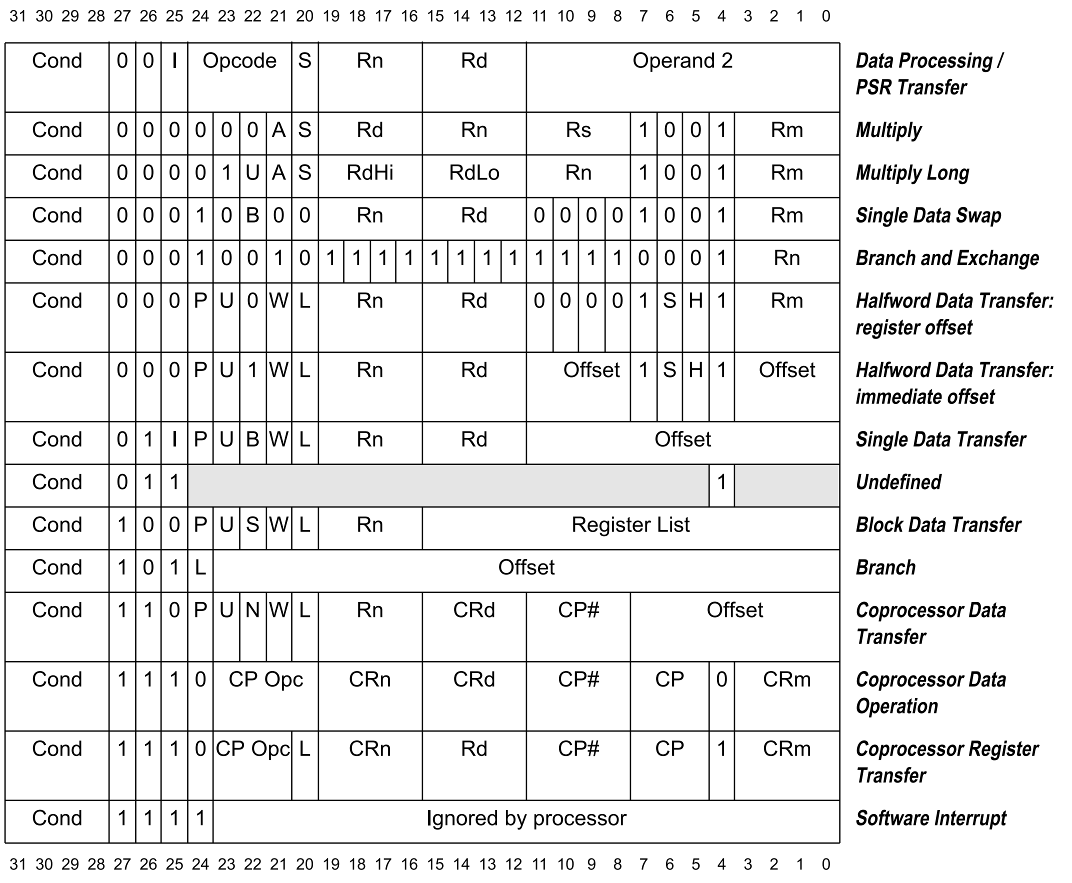
[Wright, 1979] Wright, S. (1979). *Stella Programmer's Guide*, page 2. (Cited on page 11)

[Xu, 2024] Xu, H. (2024). An ongoing attempt to re-engineer LuaJIT from scratch. Retrieved from `https://github.com/luajit-remake/luajit-remake` (accessed May 14, 2024) (Cited on page 44)

[Xu and Kjolstad, 2021] Xu, H., and Kjolstad, F. (2021). Copy-and-Patch Compilation: a fast compilation algorithm for high-level languages and bytecode. *Proceedings of the ACM on Programming Languages*, *5*(OOPSLA), 1–30. (Cited on pages 21, 43 and 44)

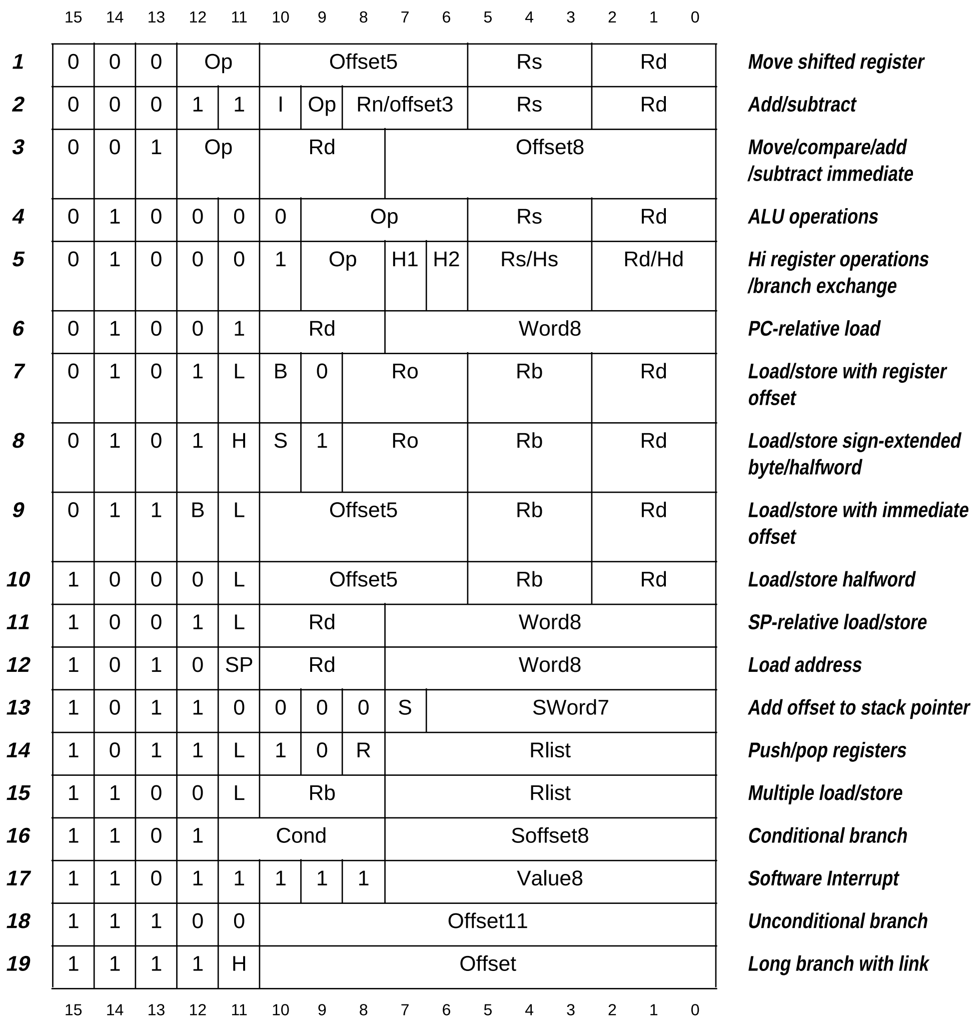
Appendix A.

Appendix



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Cond				0	0	I	Opcode				S	Rn				Rd				Operand 2												***Data Processing / PSR Transfer***
Cond				0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm				***Multiply***
Cond				0	0	0	0	1	U	A	S	RdHi				RdLo				Rn				1	0	0	1	Rm				***Multiply Long***
Cond				0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm				***Single Data Swap***
Cond				0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn				***Branch and Exchange***
Cond				0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm				***Halfword Data Transfer: register offset***
Cond				0	0	0	P	U	1	W	L	Rn				Rd				Offset				1	S	H	1	Offset				***Halfword Data Transfer: immediate offset***
Cond				0	1	I	P	U	B	W	L	Rn				Rd				Offset												***Single Data Transfer***
Cond				0	1	1	▨																				1	▨				***Undefined***
Cond				1	0	0	P	U	S	W	L	Rn				Register List																***Block Data Transfer***
Cond				1	0	1	L	Offset																								***Branch***
Cond				1	1	0	P	U	N	W	L	Rn				CRd				CP#				Offset								***Coprocessor Data Transfer***
Cond				1	1	1	0	CP Opc				CRn				CRd				CP#				CP			0	CRm				***Coprocessor Data Operation***
Cond				1	1	1	0	CP Opc			L	CRn				Rd				CP#				CP			1	CRm				***Coprocessor Register Transfer***
Cond				1	1	1	1	Ignored by processor																								***Software Interrupt***
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

Figure A.1.: ARMv4 ISA format, from [Advanced Risc Machines Ltd., 1995, p. 44].



	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	0	Op		Offset5					Rs			Rd			***Move shifted register***
2	0	0	0	1	1	I	Op	Rn/offset3			Rs			Rd			***Add/subtract***
3	0	0	1	Op		Rd			Offset8								***Move/compare/add /subtract immediate***
4	0	1	0	0	0	0	Op				Rs			Rd			***ALU operations***
5	0	1	0	0	0	1	Op		H1	H2	Rs/Hs			Rd/Hd			***Hi register operations /branch exchange***
6	0	1	0	0	1	Rd			Word8								***PC-relative load***
7	0	1	0	1	L	B	0	Ro			Rb			Rd			***Load/store with register offset***
8	0	1	0	1	H	S	1	Ro			Rb			Rd			***Load/store sign-extended byte/halfword***
9	0	1	1	B	L	Offset5					Rb			Rd			***Load/store with immediate offset***
10	1	0	0	0	L	Offset5					Rb			Rd			***Load/store halfword***
11	1	0	0	1	L	Rd			Word8								***SP-relative load/store***
12	1	0	1	0	SP	Rd			Word8								***Load address***
13	1	0	1	1	0	0	0	0	S	SWord7							***Add offset to stack pointer***
14	1	0	1	1	L	1	0	R	Rlist								***Push/pop registers***
15	1	1	0	0	L	Rb			Rlist								***Multiple load/store***
16	1	1	0	1	Cond				Soffset8								***Conditional branch***
17	1	1	0	1	1	1	1	1	Value8								***Software Interrupt***
18	1	1	1	0	0	Offset11											***Unconditional branch***
19	1	1	1	1	H	Offset											***Long branch with link***
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

Figure A.2.: THUMBv1 ISA format, from [Advanced Risc Machines Ltd., 1995, p. 108].

0x00000000 - 0x00003FFF	BIOS (System ROM)	16 KiB	General Internal Memory
0x02000000 - 0x0203FFFF	WRAM (on-board)	256 KiB	
0x03000000 - 0x03007FFF	WRAM (on-chip)	32 KiB	
0x04000000 - 0x040003FE	I/O Registers	1 KiB	
0x05000000 - 0x050003FF	BG / OBJ Palette RAM	1 KiB	Internal Display Memory
0x06000000 - 0x060017FF	Video RAM	96 KiB	
0x07000000 - 0x070003FF	OAM (Object Attribute)	1 KiB	
0x08000000 - 0x0DFFFFFF	Cartridge ROM	max. 32 MiB	External Memory
0x0E000000 - 0x0E00FFFF	Cartridge RAM	max. 64 KiB	

Figure A.3.: GBA Memory Map (without unused memory).

15 14	13	12	11 10	9	8	7 0
Shape	Color Depth	Mo- saic	Mode	Disable/ Double- Size	Rot/ Scal	Y-Coordinate

Shape: Square, Horizontal, Vertical. **Mode**: Normal, Semi-Transparent, Window.

15 14	13	12	11 9	8 0
Size	V- Flip	H- Flip	Not used*	X-Coordinate

*Bits 9 – 13 select rotation/scaling parameter set, if flag in `OBJ0` is set.

15 12	11 10	9 0
Palette Number	Priority	Tile Number

Priority is relative to background: Sprite with prio x > BG with prio x.

Figure A.4.: All OAM OBJ attributes from top to bottom (`OBJ0` - `OBJ3`).

```
signature    : "(" [paramlist] ")" ["->" retlist] [call_conv]
paramlist    : param { "," param }
retlist      : paramlist
param        : type [paramext] [paramspecial]
paramext     : "uext" | "sext"
paramspecial : "sarg" ( num ) | "sret" | "vmctx" | "stack_limit"
callconv     : "fast" | "cold" | "system_v" | "windows_fastcall"
             | "wasmtime_system_v" | "wasmtime_fastcall"
             | "apple_aarch64" | "wasmtime_apple_aarch64"
             | "probestack"
```

Listing A.1.: Full CLIF function signature grammar [bjorn3 and Singh, 2020].

Statement of Authorship

I herewith assure that I wrote the present thesis independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or to their meaning.

I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a penalty by the law enforcement agency.

Magdeburg, May 17, 2024

Signature