Introduction to Dynamic Recompilation in Emulation

Core Ideas explained within a Chip 8 context, using the x86-32 Instruction Set.

To be read in conjunction with source code available at:
https://github.com/marco9999/Super8_jitcore/tree/edu
(Super8_jitcore repo & edu branch.)

Written by:

Marco Satti (marco.satti9999@gmail.com)

Version 1.1

Contents

i. Licence 4
ii. Version History 5
iii. Terminology 6
iv. Helpful Resources 7
v. Prerequisites 9
vi. Purpose, Aim & Scope 11
vii. File Details 12
1. Introduction 15
2. Core Concepts 16
   2.1. Caches 16
   2.2. Emitter 16
   2.3. Translator 16
   2.4. Relationship Between Caches, Emitter and Translator 16
   2.5. Dispatcher Loop 17
   2.6. Interrupts 18
   2.7. Jump Table 19
3. Dynamic Recompiler – Main Structure 20
   3.1. Interpreter Structure Review 20
   3.2. Dynamic Recompiler Structure Overview 21
   3.3. Running Native Code 21
4. Implementation Issues & Details 24
   4.1. Jumps 24
      4.1.1. Jumps Problem 24
      4.1.2. Jumps Solution – Part 1 25
      4.1.3. Jumps Solution – Part 2 26
   4.2. Cache Code Generation 27
      4.2.1. Cache Code Generation Problem 27
      4.2.2. Cache Code Generation Solution 27
   4.3. Inter-cache Jumps 27
      4.3.1. Inter-cache Jumps Problem 27
      4.3.2. Inter-cache Jumps Solution 28
5. Handling Interrupts 29
   5.1. Common Details 29
   5.2. Interrupt Details: PREPARE_FOR_JUMP 29
      5.2.1. Translator Loop 30

5.3. Interrupt Details: USE_INTERPRETER 30
5.4. Interrupt Details: OUT_OF_CODE 31
5.4.1. Differences from Other Interrupts 32
6. Current Issues 33
6.1. Self Modifying Code 33
7. Conclusion 34

i. Licence

This document follows the CC BY-NC-SA 4.0 licence. See the LICENCE file which should have been included with this. You can also obtain a copy of the licence at the following website:

http://creativecommons.org/licenses/by-nc-sa/4.0/legalcode

You may find a summarised version of the licence at the following webpage. It is not a substitute for the full licence.

http://creativecommons.org/licenses/by-nc-sa/4.0/

ii. Version History

Version 1.0 (originally called Revision A) – 2016-03-07. First release.

Version 1.1 – 2016-03-11. Added in a list of files and what they are used for (File Details, Section vii). Added in Version History (Section ii). Added in text styling for better reading (see Terminology, Section iii).

iii. Terminology

Target Architecture	The machine you want to emulate (eg: Chip8).
Client Architecture	The machine the emulation will be performed on (eg: x86 processor).
Dynamic Recompilation	The process of recompiling code from the target to client architecture, which is done *during* execution.
Static Recompilation	The process of recompiling code from the target to client architecture, which is all done *before* execution.
Interpreter	The most basic form of emulation, which is magnitudes slower than recompilation methods. Involves a loop which fetches and decodes opcodes one by one.
Nibble	A single hex (base-16) code that represents half of a byte (ie: the high nibble of 0xD5 is D)
Emitter	Commonly implemented as a client helper class, this translates (human) assembly language (ie: `MOV eax, edx`) into raw bytes (`0x89`, `0xC2`).
Cache	A temporary memory block that is used to hold and execute emitted bytes (which is translated code from target to client).

Note that any assembly shown in this document is in Intel syntax, where an opcode (that permits) is followed by the destination and then followed by the source.

Throughout the document, there are different text styles to indicate the category they belong to:

Category	Style Example
C/C++ code	`emulationLoop()`
Chip8 code	`DXY0`
Interrupt code	`OUT_OF_CODE`
x86 assembly code	`JMP PTR32`

iv. Helpful Resources

The resources here relate to both dynamic recompilation and the Chip8. I recommend you read these in conjunction with this document, however it is entirely optional, as the important parts will be covered again. I would like to thank all of these people who have helped me too within these links!

Dynamic Recompilation:

Dynamic Recompilation Introduction & Building an Emitter	Article goes over some concepts of a Dynamic Recompiler and goes through the caches, emitter and translator concepts (also explained later): http://www.multigesture.net/wp-content/uploads/mirror/zenogais/Dynamic%20Recompiler.html In hindsight, this is an excellent resource (especially the 'basic components of a recompiler'), although it may be a bit confusing for beginners (and it was for me initially).
NGEMU forums: Dynamic Recompilation – An Introduction by M.I.K.e7	Good resource explaining the transition from interpreter to dynamic recompilation. http://ngemu.com/threads/dynamic-recompilation-an-introduction.20491/
PCSX2 forums: Introduction to Dynamic Recompilation by cottonvibes	Good resource from an experienced developer, from the popular PS2 emulator PCSX2: http://forums.pcsx2.net/Thread-blog-Introduction-to-Dynamic-Recompilation
Mupen64Plus Dynamic Recompiler Wiki page	Describes the inner workings of the Mupen64Plus N64 emulator: http://pandorawiki.org/Mupen64plus_dynamic_recompiler

In addition to these, you may also want to look up just-in-time compilation, which is a synonym for dynamic recompilation in the context of emulation.

Chip8:

Building an Interpretive Emulator	Article contains a step-by-step look into an interpreter emulator for the Chip8: http://www.multigesture.net/articles/how-to-write-an-emulator-chip-8-interpreter/

Specifications	Wikipedia article contains a list of the opcodes used and an explanation: https://en.wikipedia.org/wiki/CHIP-8 See also this excellent website for a more in depth look at the specs: http://devernay.free.fr/hacks/chip8/C8TECH10.HTM (may need to use Google cached copy, website is offline as of 14/11/15)
Author's own interpreter & dynamic recompiler emulators (source code)	https://github.com/marco9999/Super8 (interpreter) https://github.com/marco9999/Super8_jitcore (dynamic recompiler)

v. Prerequisites

Most of this document relies on the following things below. Please make sure you are familiar with them before continuing.

- **Assembly knowledge in x86-32.**

This is one of the main bits of knowledge you will need. The goal of this document is not to teach you about assembly or the x86 instruction set, but it will use it. A couple of great resources can be found here, which I have also used:

General info about x86 assembly: https://en.wikibooks.org/wiki/X86_Assembly

x86 instruction set reference: http://ref.x86asm.net/coder32.html

Encoding x86 instructions: http://www.c-jump.com/CIS77/CPU/x86/index.html

MSDN information about x86 architecture:

https://msdn.microsoft.com/en-us/library/windows/hardware/ff558894(v=vs.85).aspx

- **Knowledge about the target architecture (eg: Chip8 specifications)**

In order to emulate the target, you will need to know about how the target works – specification documents are your friend! Fortunately for something like the Chip8, ample documentation is freely available, could possibly be implemented based off a single Wikipedia article! See here for an example of what you might need:

Chip8 information: https://en.wikipedia.org/wiki/CHIP-8

You may not be so lucky with other target architectures however, and additional real world tests performed on the target may be needed.

- **Good C & C++ knowledge.**

Need to be familiar with the language, especially with pointers, as they will be used extensively. Although the example could be completed entirely in C, the use of classes provided though C++ is very helpful and heavily encouraged (and the language the example was done in). There are many resources on the internet to teach you about these languages through any search engine.

You may use another language if you are comfortable adapting code from C++, however it must have support for running native code at the very least (eg: possibly Java Native Interface, although I have not looked into it).

- **Calling convention knowledge.**

On Windows systems, compiling a C++ function call on Visual Studio (that is not a class member) uses the CDECL calling convention by default, which relates to how the stack is set-up. See these pages for info:

General info about calling conventions: https://en.wikipedia.org/wiki/X86_calling_conventions

MSDN article that contains useful information about registers in the x86 architecture and how they are used in calls:

https://msdn.microsoft.com/en-us/library/windows/hardware/ff561502(v=vs.85).aspx

- **How (basic) CPUs work.**

This relates to both the client and target architectures. In the case of the Chip8, it is important to know what Opcode, Program Counter (PC), etc mean. Generally, most CPUs all share some common elements. You can use a search engine to look up how these work.

- **Build an emulator using the interpreter method.**

While not strictly required, this will help you understand and gain some experience in emulation. Doing this will enable you to get more out of this document and focus on learning about dynamic recompilation. If you have not built an interpreter yet, I suggest making one for the Chip8 as it is easy.

- **A development environment**

I am using a Windows 10 x86-64 system (although we will be doing everything in 32-bit) with Visual Studio 2015 Community Edition. For graphics, sound and input you may want to use some other libraries, such as SDL, but this document does not cover that.

- **Time and willingness.**

This is quite advanced stuff! Do not expect to be an expert after reading this, or expect it to work in the first go. I encourage you to go and look at the source code of other emulators too, such as PCSX2 or Dolphin, if you are stuck on an idea. They may provide a different view or implementation.

vi. Purpose, Aim & Scope

In terms of dynamic recompilation, by the end of this document you should be familiar with:

1. What dynamic recompilation is and the advantages over a traditional interpreter approach.
2. How to handle problems involved in dynamic recompilation based around a Chip8 context.
3. The basic structure involved in dynamic recompilation, again based around a Chip8 context.

In addition, example source code where relevant will be provided based on the Chip8 system. You will need to consult the Super8_jitcore repo on my GitHub (marco9999) account for the full source code (the *edu* branch should be consulted). Most of the code provided in here is where concepts need to be explained a bit more.

This document does NOT provide the following (also look at Prerequisites):

1. Teach you about the Chip8. While it is a definitely a non-complex system to learn about, it does not go in depth at all about the system, and it is assumed you can follow what is happening using examples though a Chip8 context.
2. Teach you about the interpretation method. While it does recap the process, it does not explain the elements or go into depth. Making sure you understand this will set you up to understand the rest of this document.
3. C++ or Assembly explanations. It only uses the languages to code the Chip8 example in. This also includes how to encode x86 instructions in raw bytes.
4. Terminology explanations. Please look up in a search anything you do not understand, as it is most likely important. If you don't understand for example what an opcode is, go do some research before starting. Some of the more important ones have been provided at the start.
5. Optimisations. There is probably lots of opportunity to do optimisations in the source code, but it will be focused on being helpful rather than on speed.
6. A comparison to other emulators. Although some core concepts might be the same, the implementations may be vastly different.
7. Setting up a development environment or anything related to it.

And most importantly:

8. This document DOES NOT COVER EVERYTHING! There are MANY different ways to implement a dynamic recompiler – this is just one of them. Take time to research other emulators if you are stuck on a problem as the answer might already be there.

vii. File Details

This section lists the files included under the Super8_jitcore/edu repo, including brief descriptions for each. All files are contained in the Super8_jitcore/ folder

File	Description
Headers/Globals.h Source/Globals.cpp	Global header that contains debugging toggles used to increase logging frequency. Also contains the `USE_SDL` define which specifies if graphics output using the SDL library is to be used. Source file is not of importance.
Headers/SDLGlobals.h Source/SDLGlobals.cpp	Contains SDL global variables used throughout the emulator. If the `USE_SDL` flag is defined (in Globals.h), the `SDL_gfxmem` and `SDL_pitch` is used when drawing Chip8 sprites (used by the `Interpreter` class), and will update the SDL window created.
Headers/Super8.h Source/Super8.cpp	Header file is empty, not used for anything. Source file contains the main program loop, including updating the graphics output. Can be thought of as a `while(1)` loop that executes the `emulationLoop()` function of the `MainEngine` class.
Headers/Chip8Engine/CacheHandler.h Source/Chip8Engine/CacheHandler.cpp	Header and source file for the `CacheHandler` class. The cache handler is responsible for allocating and managing cache memory, and the setup CDECL cache. `CacheHandler` coordinates with the `JumpHandler` class to invalidate caches/jumps.
Headers/Chip8Engine/CodeEmitter_x86.h Source/Chip8Engine/CodeEmitter_x86.cpp Source/Chip8Engine/CodeEmitter_x86_ADD.cpp Source/Chip8Engine/CodeEmitter_x86_Bitwise.cpp Source/Chip8Engine/CodeEmitter_x86_Jump.cpp Source/Chip8Engine/CodeEmitter_x86_MOV.cpp Source/Chip8Engine/CodeEmitter_x86_SUB.cpp	Header and source files for the `CodeEmitter_x86` class. The `CodeEmitter_x86` class is responsible for the emitter portion of the dynamic recompiler. It contains functions that when used write the raw bytes into the selected cache. Specialised opcodes (including `DYNAREC_EMIT_INTERRUPT`) are implemented in the reference source file, while the other source files contain the relevant specific opcodes.

Headers/Chip8Engine/Interpreter.h Source/Chip8Engine/Interpreter.cpp	Header and source file for the `Interpreter` class, which is used for any opcodes that are not implemented in the translator. Only the draw opcodes are contained in the interpreter, the others are not needed and commented out.
Headers/Chip8Engine/JumpHandler.h Source/Chip8Engine/JumpHandler.cpp	Header and source file for the `JumpHandler` class, which implements the jump table concept mentioned in this document. Coordinates with the `CacheHandler` to allocate new caches when needed.
Headers/Chip8Engine/Key.h Source/Chip8Engine/Key.cpp	Header and source file of the `Key` class, which implements the keypad used with the Chip8.
Headers/Chip8Engine/MainEngine.h Source/Chip8Engine/MainEngine.cpp	Header and source file of the `MainEngine` class, essentially the entry point for this emulator. Implements the dynamic recompiler structure, including the dispatcher loop, executing caches, plus handling interrupts. Implements part of the translator loop, used in conjunction with the Translator class.
Headers/Chip8Engine/StackHandler.h Source/Chip8Engine/StackHandler.cpp	Header and source file of the `StackHandler` class, used whenever stack jumps are encountered.
Headers/Chip8Engine/Timers.h Source/Chip8Engine/Timers.cpp	Header and source file of the `Timers` class, used to implement the Chip8's sound and delay timer. In the edu branch, it does NOT count down at 60Hz, and instead counts down once per translated instruction (see the Translater class source code).
Headers/Chip8Engine/Translator.h Source/Chip8Engine/Translator.cpp	Header and source file of the `Translator` class, which implements part of the translator loop. The translator loop is designed to translate code for a block of Chip8 code, which generally means until a jump occurs.
Headers/Chip8Globals/C8_STATE.h Source/Chip8Globals/C8_STATE.cpp	Contains the Chip8 hardware implementation, such as the CPU registers, memory among other things. Used with both the recompiled code as well as the `Translator` class.

Headers/Chip8Globals/MainEngineGlobals.h Source/Chip8Globals/MainEngineGlobals.cpp	Contains global variables used between components. The draw variable is also set whenever a draw has been done and the SDL window needs to be updated.
Headers/Chip8Globals/TranslatorGlobals.h Source/Chip8Globals/TranslatorGlobals.cpp	Contains a global variable used between the `Translator` class and translator loop function in the `MainEngine` class.
Headers/Chip8Globals/X86_STATE.h Source/Chip8Globals/X86_STATE.cpp	Contains global variables related to the x86 execution state of the emulator (ie: when the translated caches are run). Includes the resume emulation address and parameters used when an interrupt is raised.
Headers/Logger/* Source/Logger/*	External component (not relevant to emulator).

1. Introduction

The dynamic recompilation method means to re-compile during runtime the target program machine code into the client machine code. When compared to the more basic interpretation method, they both share the principle of having to grab an opcode, decode it and perform the function. The key difference between the two however is that dynamic recompilation keeps the translated code in memory regions known as caches, which are recalled on an on-demand basis, whereas interpreters may perform the same opcode over and over, having to translate the opcode each time.

Dynamic recompilation is used vastly in emulation, mostly for its end benefits. Compared to the basic interpretive emulation it offers very large improvements but also introduces problems which are more difficult to tackle. Benefits of a dynamic recompiler include:

- Magnitudes faster than a basic interpreter (speed or power saving benefits):
  There is no need to translate every opcode on every cycle, as the result has already been cached before. This means the emulator can just point to the cache and run the code, which means a large speed increase.

That's it! There is really no other reason why you should use a dynamic recompiler over an interpreter. If you do not need the speed increase, you are just wasting your time (unless you want a challenge), as it introduces an incredible amount of complexity to the emulator, such as:

- Much harder to debug:
  Often you will not know where a problem is even if your code looks ok. It could be as simple as using the wrong emitter function bit-ness (ie: using 8-bit instead of 16-bit), or a problem that only arises once compiler optimisations are turned on. Usually there is also a much larger code base you will need to maintain, which has a higher chance to be bug-prone.

- Problems relating to jump locations:
  It is usual for architectures to employ jump instructions for program flow control. This creates a problem on a client architecture such as the x86 where instructions can be anywhere in length up to 16 bytes, meaning you do not know where you should jump to.

- Problems with timings/synchronisation between components:
  In a simple system such as the Chip8, there is not much of a problem here. When you start to look at a complex system however, such as one with a GPU, SPU, etc you will quickly run into problems trying to synchronise data or with a system that relies on accurate timings.

This document attempts to give you understanding of how a basic dynamic recompiler look like in terms of explanations, diagrams and source code, and provides solutions to the problems above and more.

2. Core Concepts

This section provides the foundation ideas upon which a dynamic recompiler is built upon within emulation. All of these concepts will be used with building an emulator, so make sure you understand them before moving on.

2.1. Caches

Caches are at the heart of a dynamic recompiler, where they are used to read, write and execute translated code. Without these caches, the dynamic recompiler will not work – no exceptions. While some emulators implement caches differently, the core idea is the always the same and that is to hold the translated code within a memory block so it can be read again for execution.

The client (x86) emitter class is usually going to be the only class that will directly write to a cache (Section 2.2) and the code will be read from the main program via a call to the entry point.

More often than not, a cache will hold translated code until a branch is hit (ie: a jump in code). This is done in order to solve a specific problem, which is discussed in Section 4.

In my emulator, I allow any number of caches to be created, in order to hold the translated Chip8 code. While this might sound like a bad practice, the size of Chip8 memory is only 0xFFF (4095 bytes) in size, so not much memory is taken up by the total number of caches. Again your implementation can dictate the parameters and structure – read up on some of the other emulators for examples.

2.2. Emitter

The emitter class is a helper class that is used to construct client assembly instructions through the use of functions. When translating code, it would be very tedious if every time you manually committed bytes to a cache to represent instructions. Instead, the emitter class is used to do the hard work for you.

For example, a common assembly instruction is the `MOV` instruction, but it can take 2 parameters of which they can be specific combinations of intermediates, memory address or registers. The emitter class would contain functions to emit bytes based on all of these specific combinations.

While from a technical point of view, it would be possible to not include this class, it would not be wise to make a dynamic recompiler without one – I can't emphasise this enough! I have included it as a critical part of the program as it is just too hard to make an emulator without this class (at least for me).

2.3. Translator

The translator class & functions takes on the role of converting target opcodes into client opcodes (or instructions). This is largely similar to an interpreter cycle, but the key difference here is instead of immediately performing the action, it uses the emitter class to store the instructions to be executed later in a specified cache.

2.4. Relationship Between Caches, Emitter and Translator

I came across a webpage which further explores the relationship between these 3 components – the caches, emitter and translator. It also contains a basic structure diagram which I found helpful while learning about it. See here (from Section iii):

http://www.multigesture.net/wp-content/uploads/mirror/zenogais/Dynamic%20Recompiler.html

At first I didn't really understand it, but as I researched and developed more code I began to see what the author was trying to say.

In essence, this is what the structure looks like, adapted from the diagram given from that webpage (thanks to the author *zenogais*!):

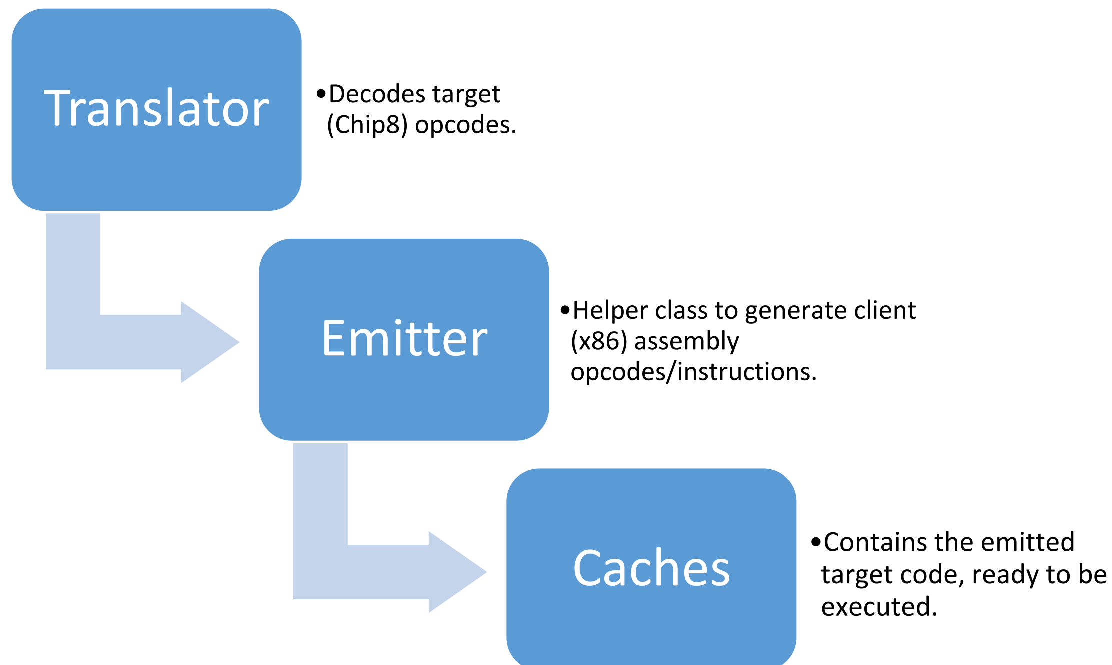

Figure 1: Relationship between the translator, emitter and caches

Note: this is a least critical (top) to most critical (bottom) diagram, whereas *zenogais* describes his diagram as bottom to top. They represent the same thing. Without caches however, the most critical part of the program is missing. This is what *zenogais* was trying to achieve with their diagram (order of importance).

This diagram essentially demonstrates the relationship and order of use between the 3 components, in a simplified manner. If any of these 3 components are not present in the program, it will not work – it's as simple as that!

How these components are implemented is briefly discussed in Section 5.2.1. See the source code for the full implementation.

2.5. Dispatcher Loop

The meaning of the dispatcher loop (main program loop) here is twofold:

1. Interface to the underlying operating system, which is used to update user interaction elements (graphics, sound, input, etc).
2. Initiate execution of the caches, and act as a handler for the interrupts generated.

This loop is abstracted over the top of all components in the emulator, tying everything together. It also essentially acts as the entry point into the emulator.

Most of the code shown in this document for the dispatcher loop will only be for point 2 – implementing the user interaction code will be up to you (I suggest looking at SDL if you are interested, or the master branch of my repo).

2.6. Interrupts

In the realm of computer science, interrupts are a signal that is generated that is used to get the immediate attention of a component, which can then be serviced. In this case, the component is the dispatcher loop. Interrupts are useful within dynamic recompilation as it provides a way to service runtime problems before executing code – such as jumps mentioned in the introduction.

Within my emulator, an interrupt will be raised inside a translated code cache, which will cause the program to transfer control outside to the dispatcher loop. The interrupt is then serviced by the appropriate handler, and transfers control back to the translated code. The following is an example of what an interrupt will cause:

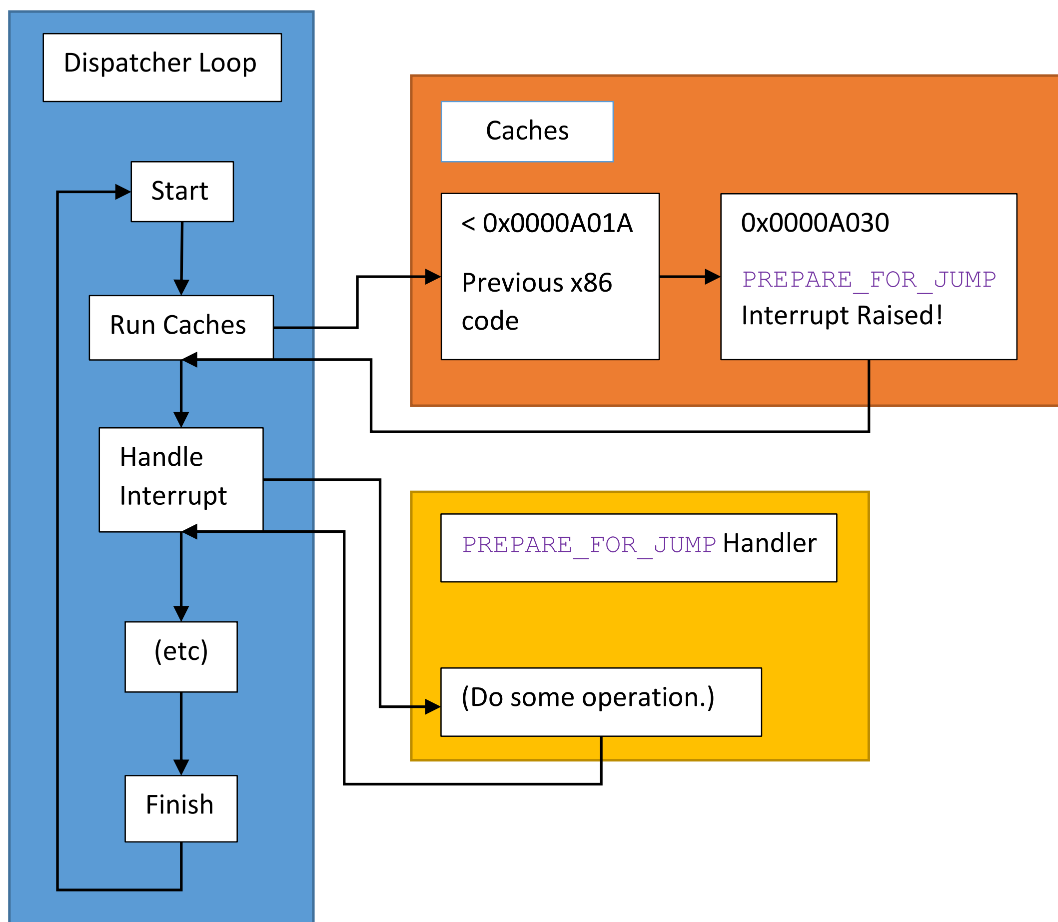

Figure 2: Visual representation of the interrupt handling process.

More information will be provided throughout the document where the dynamic recompiler structure and various interrupts will be discussed. Only the most used interrupt codes will be explained, detailing what they are used for and a summarised version of how they work. Once you are familiar with how these work, the others will make sense by reading the source code and comments.

From a technical point of view, while these interrupts could be serviced inside the translated code directly, it is much easier to handle them inside the dispatcher loop using C++. This causes a little bit of overhead to be introduced but it is not a problem for this emulator. Take a look at the `DYNAREC_EMIT_INTERRUPT` function later on inside the `CodeEmitter_x86` class to see how interrupts are generated within caches.

2.7. Jump Table

A jump table within an emulation context is a mapping between two memory addresses (between target and client memory), which are used to lookup where a jump should go when performed on the client architecture. In general, whenever a jump is encountered it will reference an entry in the jump table within the emitted translated code. Visually, the table will look something like this:

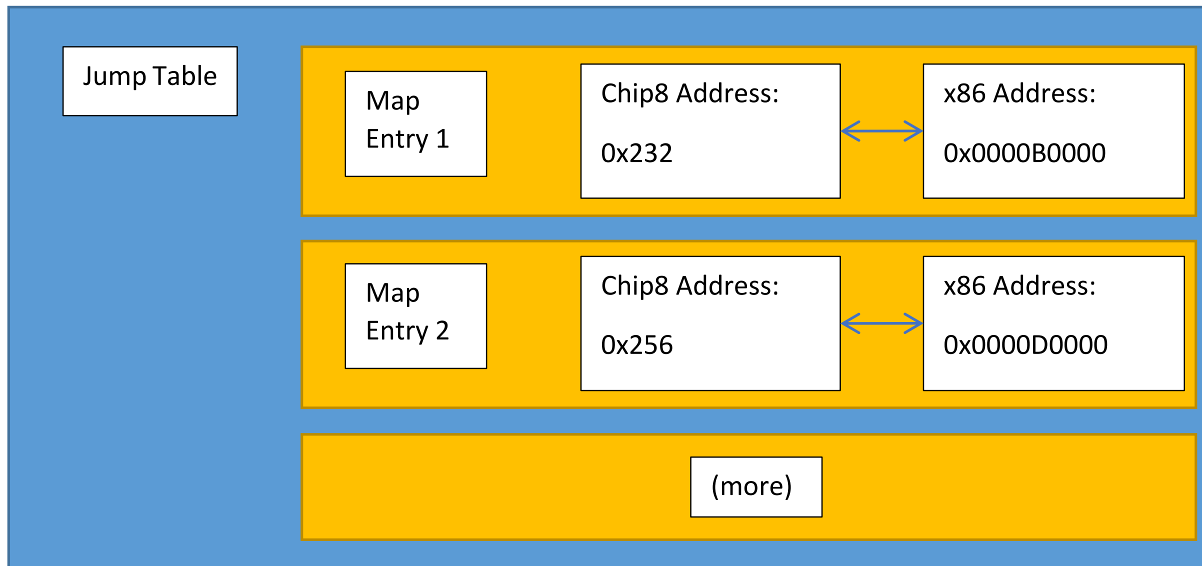

Figure 3: Jump table visual representation, mapping Chip8 addresses to x86 addresses.

3. Dynamic Recompiler – Main Structure

3.1. Interpreter Structure Review

Before introducing the dynamic recompiler structure, I wanted to recap what is involved in the interpreter emulation structure, to help draw better comparisons between the two. For those that have built an interpreter, this structure will be immediately familiar. This style of emulation attempts to replicate at a software level exactly how the machine would work. This diagram is shown with a Chip8 context, but most of the elements are similar. If you are stuck on any element of this diagram, please go and review an emulator which uses the interpreter method.

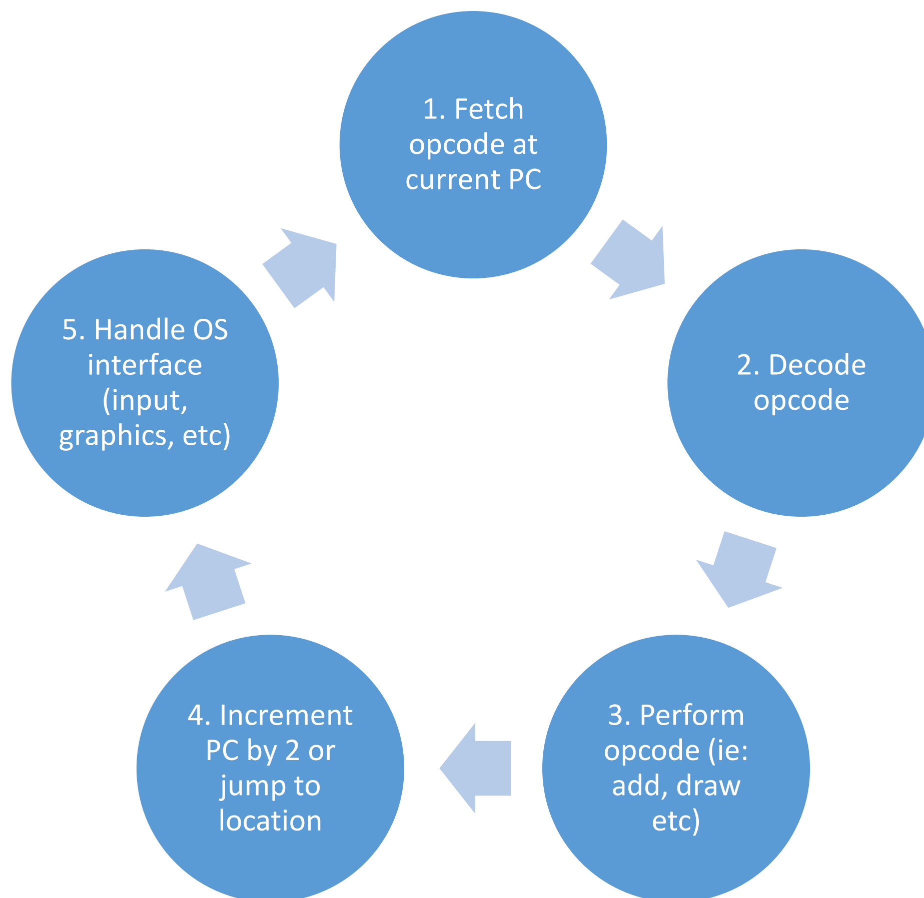

Figure 4: Interpreter structure overview.

3.2. Dynamic Recompiler Structure Overview

The dynamic recompiler structure deviates from the interpreter structure, where it shifts the **translator loop** (which is basically the diagram above) away from being the main loop. Instead, the main loop, alternatively called the **dispatcher loop**, looks conceptually like this:

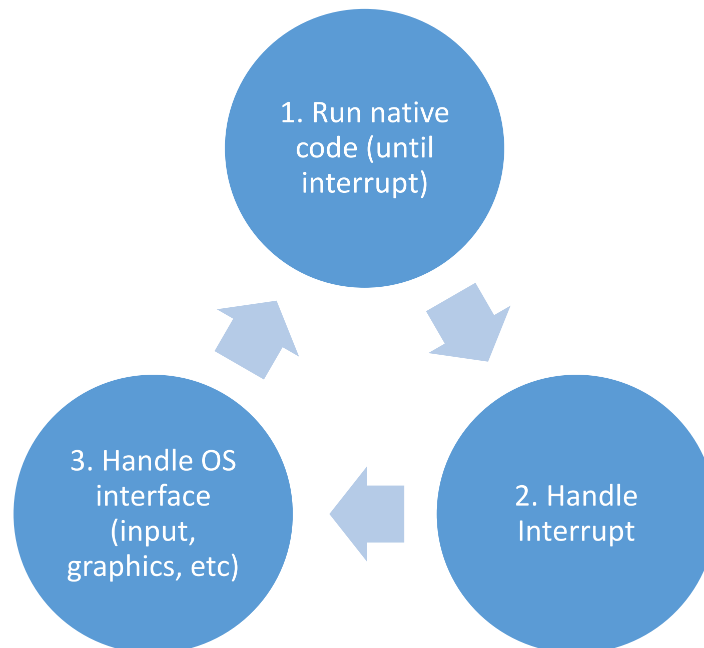

Figure 5: Dynamic recompiler structure overview.

At the core that's all there is to it… so you're probably wondering where all the complexity comes from, and the answer lies in handling interrupts (which includes translating code). With the interpreter structure in the section above, the emulator is basically complete with only that diagram. With the dynamic recompiler structure however, the interrupt flow block needs its whole sub-diagram, which is explained further on throughout. For now, only part 1 is discussed below, as we first need to explain where some interrupts come from in Section 4.

This setup can be found in the source code through the `MainEngine::emulationLoop()` function (for 1 and 2), as well as the `main()` function (for 3 and the whole loop).

Note that this setup is probably not the best way to implement a dynamic recompiler emulator – it wouldn't be a very good emulator for example if the input was only updated after a block of code was executed! For demonstration purposes this works well and is used.

3.3. Running Native Code

Looking at part 1 of the dynamic recompiler structure, running native code until an interrupt is raised, this is where we run the translated code from the target architecture into the client architecture – ie: running x86 code in place of Chip8 code.

In order to start the execution of translated code, we must know where to start executing from. Initially, this will be at the start of the cache that contains 0x200 as the start C8 PC – this is the normal entry point for a Chip8 program. Eventually however, we will also need to handle interrupts, which requires resuming emulation from a different x86 memory address (within a cache), such as when a jump happens. This is where a global pointer `x86_resume_address` is introduced – this pointer

variable stores the address at which emulation will resume. It is located within the `Chip8Globals::X86_STATE` scope.

Whenever an interrupt is generated through `DYNAREC_EMIT_INTERRUPT` (located in the `CodeEmitter_x86` class) or otherwise, there is x86 assembly code that stores the current x86 `EIP` address in this variable, before transferring control back to the dispatcher loop (special case for the `OUT_OF_CODE` interrupt, explained later). This makes sure that upon the dispatcher loop finishing up handling interrupts, it will resume emulation at the point at which it was previously stopped. With this in place, we are now able to start and stop emulation at any time.

There is one other piece to running the native code – the CDECL calling convention. Initially my attempts to create an emulator relied on function pointers to start executing native code. When a function call is made under Windows/Visual Studio compiler, by default it uses the CDECL calling convention to call a function. I have stuck with this convention even though it is sufficient just to use a simple JMP instruction (as the stack is not modified during execution).

In order to support this method of function calling, a small cache called the “setup CDECL cache” is created at runtime which handles the stack frame creation, jump to the emulation resume point (`x86_resume_address`) and stack frame destruction. Consult the `execCache_CDECL()` and `setupCache_CDECL()` functions within the `CacheHandler` class for more information. Visually, this whole process of executing the caches is displayed below.

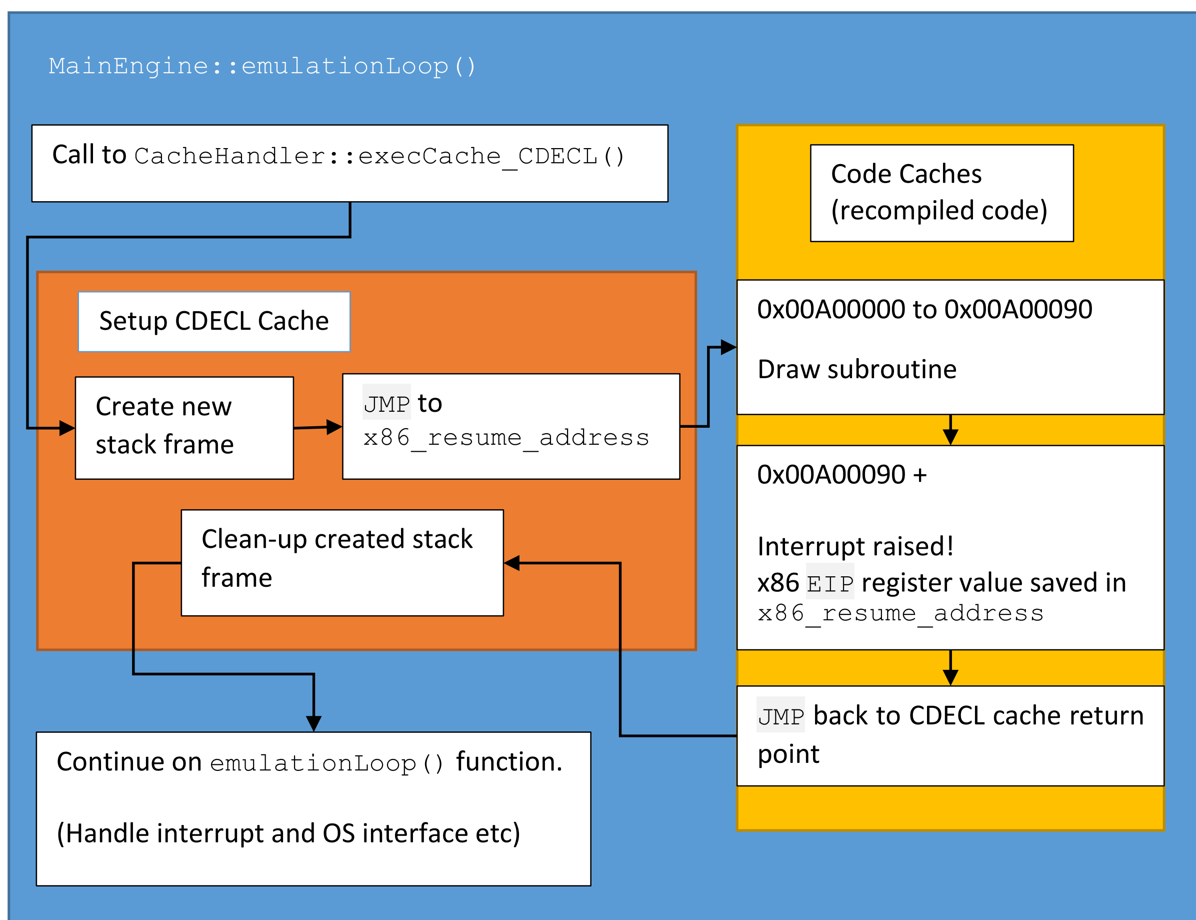

Figure 6: Diagram of cache execution using the CDECL calling convention.

There is one other part of the CDECL cache I should explain from the source code, and that is about the `EIP` value 'hack'. The x86-32 instruction set contains no opcode to directly read the `EIP` register. Normally this would be possible with mostly any other register through the MOD-REG-R/M byte and a move instruction, however the bit sequence for the `EIP` register simply does not exist. To get around this (as we need the `EIP` address to resume emulation), a call is made to the `CacheHandler::setup_cache_eip_hack` address from a parent function, which stores the `EIP` address on top of the stack. This is then pop'd off the stack into the `EAX` register (and of course push'd again), which is kept across the return opcode. The parent function that called the hack method can then use the `EIP` value that is kept within the `EAX` register.

4. Implementation Issues & Details

Before diving into interrupt details for a dynamic recompiler, we will need to discuss the technical problems associated with this method. For a system such as the Chip8, there are not many issues around dynamic recompilation. In fact, most of the problems listed will be common to all emulators, not just the Chip8 system. After we have discussed these problems, we will be ready to put the emulator together.

4.1. Jumps

4.1.1. Jumps Problem

With the interpreter approach, handling jumps was easy. We would just point the Chip8 PC to the jump location and re-run the whole emulation loop. This method works because we do not care where the jump points to – we will always know the address it points to and be able to run code from that point. It is important to note that this is always done in the target (Chip8) context. Here is an illustration of this in the Chip8 context:

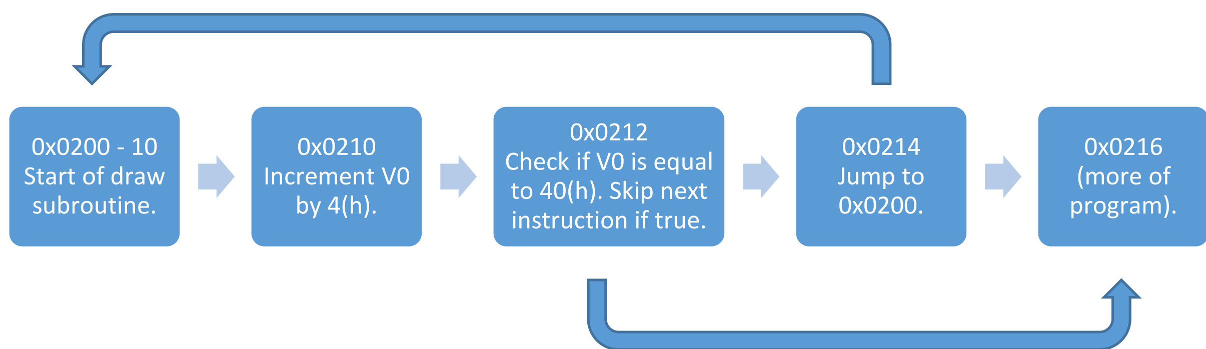

Figure 7: Chip8 example of jumps.

With dynamic recompilation, we are now working within the client (x86) context. All of a sudden, we may encounter a translated instruction where we have to jump back into code that we have already translated. This creates a problem, as on an architecture such as the x86, where opcodes vary in length, and we do not know where the jump should go to as the future opcode has not been translated yet. Here is an illustration of this problem in the x86 context (remember we are working in 32-bit addressing mode, and no longer within the Chip8 context):

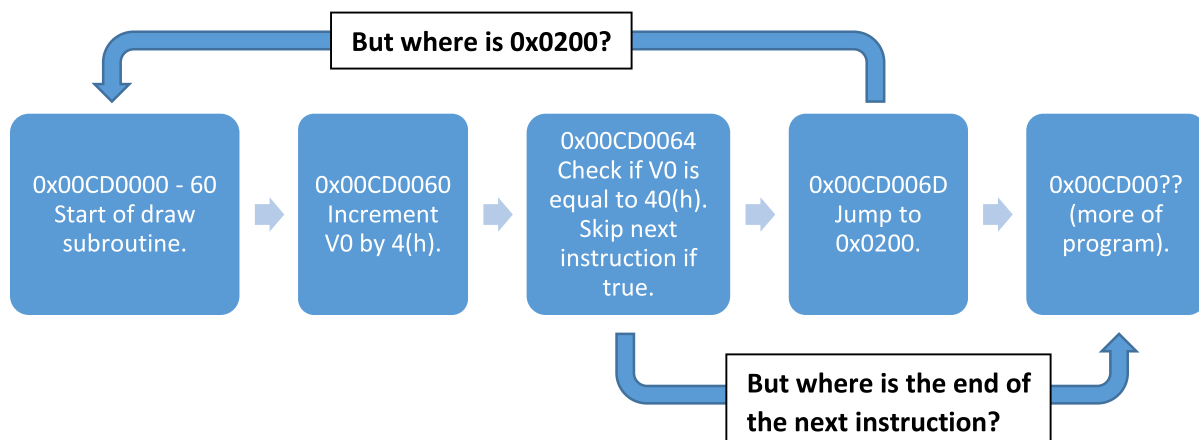

Figure 8: Non-working x86 representation of Chip8 jumps, highlighting problems.

Clearly there is a big problem here – if we do not know where to jump to, then our recompiled program will not work at all if it contains jumps!

Now is a good time to introduce the types of jumps we will encounter in a Chip8 context. Not all of them are the same, and we will have to handle them in different ways.

Direct Jumps

From my own experience, these are probably the most widely used jump type in a Chip8 context, besides conditional jumps. The only Chip8 opcode that falls under this type of jump is `1NNN`, where it points the PC to the address `0xNNN`.

Indirect Jumps

This type of jump is almost never used in a Chip8 context. In spite of this, it is important to recognise that these types of jumps can only be determined during runtime - ie: you can't determine where the jump will go to until you reach this instruction in the program. This is because indirect jumps depend on the state of the CPU while calculating where to jump to. There is only one opcode of this type in the Chip8 context, and that is `BNNN`, where it jumps to address `0xNNN + V0` – as you can see it depends on the value within the `V0` register before it can determine the jump location.

Stack Jumps

Often used to handle sub-routine calls, stack jumps are where it puts the return location onto the stack and jumps to another address (that is known pre-runtime – like a direct jump). Once done with the subroutine, a return call is made which jumps back to the address at the top of the stack (but can't be determined pre-runtime). Within the Chip8 context, two opcodes are used for stack jumps: `2NNN` and `00EE`, where the first is for calling a subroutine, and the second is for returning from a subroutine.

Conditional Jumps

The other widely used jump opcode in a Chip8 context, conditional jumps are used a lot for program logic flow. The Chip8 opcodes that fall under this category are `3XNN`, `4XNN`, `5XY0`, `9XY0`, `EX9E` and `EXA1` where they all skip the next instruction if the condition is met. We can use the fact that they only skip *one* Chip8 instruction to develop a solution for this type of jump.

Reflecting on these types of jumps, one could actually categorise them based on two major categories: independent (compile time) or dependant (runtime) jumps. However, we will leave them as they are, as I have made solutions for each of the 4 categories.

4.1.2. Jumps Solution – Part 1

Direct Jumps, Indirect Jumps & Stack Jumps

In order to support these jumps in the first place, we need a way to determine which x86 address corresponds to which Chip8 jump address.

This is where the idea of segmented (multiple) caches comes in – creating caches that contain translated code from the beginning of a specific Chip8 address corresponding to jump locations, and end on a Chip8 address location when a jump instruction is reached. From now on, this is what a single cache refers to and is also known as a memory block – a continuous block of instructions that does not branch until the last instruction in the cache.

At this point, this is an example of what the cache layout would look like if we have a rom loaded and took a look, remembering & assuming these things:

- Assume we have a Chip8 rom loaded, and we have translated code from Chip8 memory locations 0x0200 – 0x0260 (but there is additional code beyond 0x0260 not shown).
- Remembering that Chip8 opcodes take up 2 bytes of space, but the translated x86 opcodes can vary in length (and are usually more than 2 bytes long). The number of bytes the x86 opcodes take up is represented by the x86_pc variable within each cache.
- The caches have been allocated in x86 with sufficient space to hold the translated code and starts at 0xNNNNNNNN.
- The cache start and end Chip8 PC are INCLUSIVE (ie: translated code exists in the same cache for the boundary Chip8 PC's and anything in-between).
- Assume that the start Chip8 PC's are jump locations, and the end Chip8 PC's are jump instructions (which all contain known Chip8 jump addresses).

Figure 9: Example of a cache layout.

Ok great – we now have a basis to deal with jump locations. When a translated x86 jump instruction is reached, we can simply jump to the corresponding start x86 address of a matching cache which has the same start Chip8 address.

In order to support self-modifying code, the jump table concept mentioned earlier is implemented in the emulator. In terms of the cache instructions when a jump is performed, it is necessary to first perform a `PREPARE_FOR_JUMP` (or similar) interrupt, and then make use of the `JMP PTR32` instruction. The `PREPARE_FOR_JUMP` interrupt makes sure that there is a valid cache to jump to (explained in more detail in Section 5.2), and the `JMP PTR32` opcode is used instead of a standard `JMP IMM32` so it can jump to the address contained in the jump table, allowing for different jump locations.

4.1.3. Jumps Solution – Part 2

Conditional Jumps

Conditional jumps are somewhat different than the other types of jumps, where in the context of the Chip8 system, it will always skip the next instruction if the conditions are met – ie: we know the relative jump distance is forward 1 Chip8 instruction.

This means that if a conditional jump is encountered in the translator loop, we can do the following:

1. Emit the x86 `JMP REL32` instruction as per normal, but do not fill in the relative value.

2. Record the cache address minus 4 for the relative jump value address. This is assuming we are using a 32-bit x86 relative jump instruction such as `JE 0x00000000`, where 4 bytes are used for holding the relative jump value.
3. Keep running the translator.
4. When one translator cycle has passed, we can go back to the relative jump instruction and fill in the relative value with the current cache address minus the cache address recorded above. This will complete the relative jump instruction and make it valid.

4.2. Cache Code Generation

4.2.1. Cache Code Generation Problem

As it is not possible to generate the x86 code all at once from the Chip8, a decision needs to be made in regards to when the x86 code should be generated.

4.2.2. Cache Code Generation Solution

The solution implemented is to generate the code when a `PREPARE_FOR_JUMP` interrupt is handled. As one of the interrupt parameters is the Chip8 jump location, we are able to grab the details of the corresponding jump table cache, and check if the cache `x86_pc` is equal to 0 to determine if code is to be generated. In other words, this is done on an on demand basis and not all caches will have code generated at the same time.

This is also implemented in the `PREPARE_FOR_STACK_JUMP` and `PREPARE_FOR_INDIRECT_JUMP` interrupt handlers, with mostly the same method.

Once the correct cache has been selected and switched to in the `CacheHandler`, the translator loop is run up until the next jump (hence translating a complete block of code).

4.3. Inter-cache Jumps

4.3.1. Inter-cache Jumps Problem

Hopefully you'll remember that caches should always end on a normal x86 jump instruction from the general jumps problem. A problem that arises from the conditional jump solution mentioned above is that often there are loop constructs that look like the figure below, where even though the cache does end on a jump, there is no translated code to execute if the conditional jump is true.

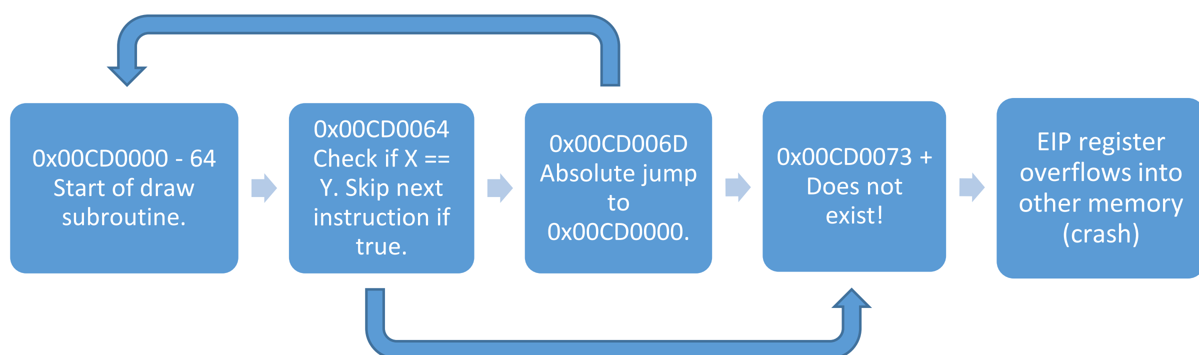

Figure 10: Example of the inter-cache-jump problem which stems from conditional jumps.

As you can see, the conditional jump instruction will have nowhere to go if true, even though the code is valid. This is because a different cache will contain the code that goes after the end instruction, which was created due to the 'end on jump' cache rule mentioned in Section 4.1.2.

4.3.2. Inter-cache Jumps Solution

To solve this, we need to emit a jump to another cache, which will contain the code after the cache end jump instruction. We will know when to do this when an `OUT_OF_CODE` interrupt is raised, which is placed at the very end of the cache at creation (as a guard). The rest of the cache is filled with `NOP`'s at creation, which means any unused space will simply do nothing while reaching the interrupt.

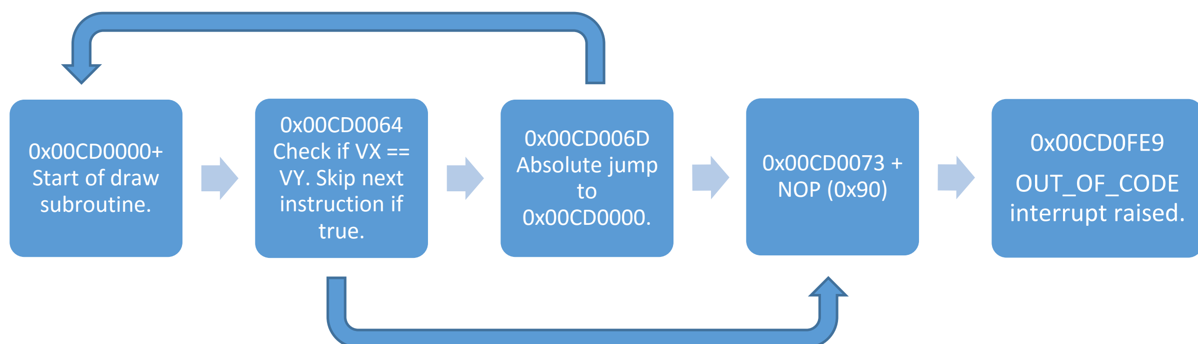

Figure 11: Diagram of the inter-cache jump solution with the guard at the end of the cache.

The method of jumping is the same method used in the general jumps problem above – that is to use the jump table and emit a `JMP PTR32` x86 instruction. The emulation resume address is also set to the last address in the cache where code was emitted, so the jump can happen.

See Section 5.4 for more information on the `OUT_OF_CODE` interrupt.

5. Handling Interrupts

This section will go over the different interrupts that will be encountered while running the emulator. Not all of the interrupts will be covered, only the essential ones. Once you learn how some of the major interrupts work you should be ok with decoding the other minor interrupts not covered.

The major interrupts that are commonly used throughout Chip8 emulation are:

1. `PREPARE_FOR_JUMP`
2. `USE_INTERPRETER`
3. `OUT_OF_CODE`

5.1. Common Details

All of the interrupts involved have access to 3 parameters, which are used to parse information when the interrupt gets handled later on. Of these 3 parameters, currently two 16-bit parameters are used to hold Chip8 addresses (ie: 0x210) or opcodes and one is used to hold an x86 32-bit address (ie: 0x00BA0000). Usually at least one of the parameters are used for holding information by an interrupt.

5.2. Interrupt Details: PREPARE_FOR_JUMP

The `PREPARE_FOR_JUMP` interrupt is used to determine where a jump should go to before performing the jump. Within the translated code cache that this interrupt is in, an emitted `JMP PTR32` x86 instruction will be performed through the jump table after this interrupt is serviced.

One may think that it is ok to use a static `JMP IMM32` instruction. However, there is one major problem with this, and that is that self modifying code would not be supported. In terms of Chip8 opcodes, this means anything that writes back to memory, such as the `FX33` and `FX55` opcodes, would be affected. Every time these opcodes are executed, the caches representing the written memory are no longer valid and do not represent the current memory of the Chip8 system. When a cache is marked invalid (through the `SELF_MODIFYING_CODE` interrupt), it is scheduled for deletion as soon as possible.

To resolve this, a jump table is created, which involves holding all of the Chip8 jump locations in a table, individually mapped to a x86 address value that can be dynamically updated in the `PREPARE_FOR_JUMP` interrupt. When a `JMP PTR32` instruction is emitted, it points to the x86 address variable in the jump table which contains the correct jump location.

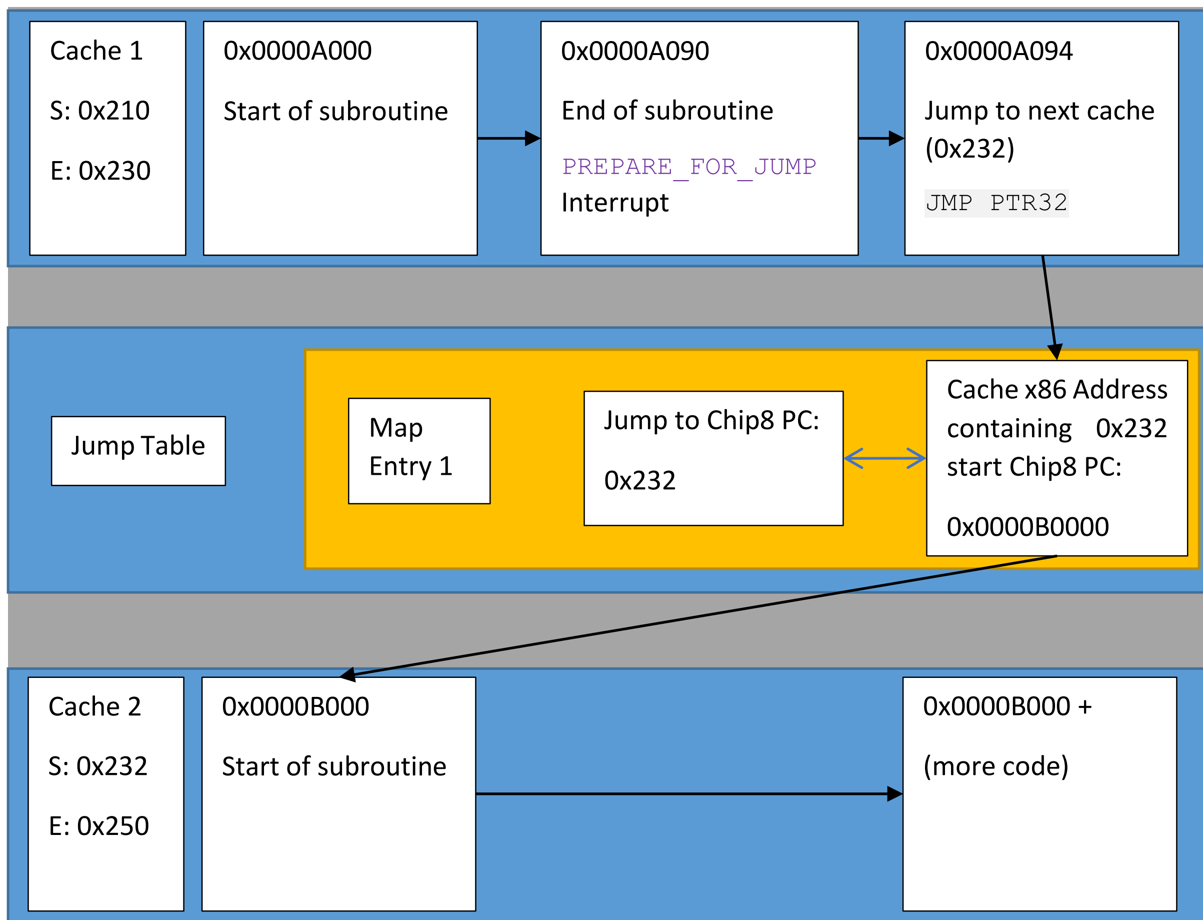

Figure 12: Diagram to show how jumps interact with the jump table and between caches.

If a cache has previously been deleted, the `PREPARE_FOR_JUMP` interrupt makes sure there is a cache to jump to by creating one and updating the jump table entry.

As mentioned before, there is one other purpose of the `PREPARE_FOR_JUMP` interrupt, and that is to initiate the translator loop on caches that contain no code. One of the parameters parsed to the interrupt handler is the Chip8 jump location. The handler checks the corresponding cache in the jump table through this parameter, to see if any code has been emitted before by checking the `x86_pc` cache variable. If no code exists, the translator loop will initiate and translate a block of code.

5.2.1. Translator Loop

The translator loop is responsible for translating the target opcodes (Chip8) into the client opcodes (x86). This is similar to how the interpreter emulator works, where it replicates how the Chip8 CPU works. However instead of executing the opcode, it merely stores the translated equivalent in a cache for later execution.

The translator loop is designed to run for a block of code, meaning until a jump is reached (normal jump, stack jump or indirect jump).

5.3. Interrupt Details: USE_INTERPRETER

Some opcodes within the Chip8 specifications are complex to translate directly into x86 emitted assembly, such as the `DXY0` opcode (draw sprite). For these opcodes, the implementation from the interpreter approach is reused (ie: using C++ code), and the dynamic recompiler approach can be developed later.

This approach only works if the Chip8 CPU state is synced beforehand, meaning that all of the registers and memory have to be updated before the interpreter can be used. In terms of emulation, this means running the translated code up until the interpreted opcode.

In order for the program control to fall back to the interpreter, an interrupt is emitted at the opcode position in the cache with the `USE_INTERPRETER` code (and the Chip8 opcode as the parameter). This signals the dispatcher loop to use the interpreter to handle the opcode.

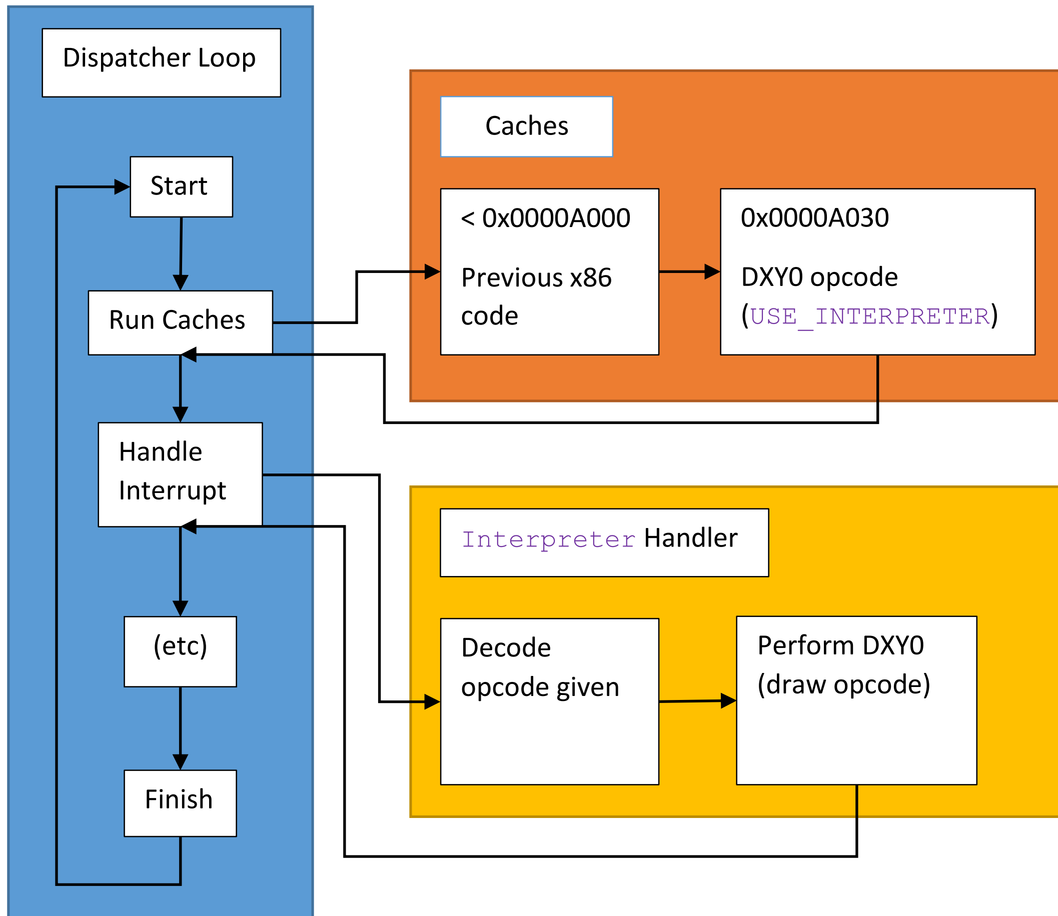

Figure 13: Flow diagram of the interpreter fall-back process.

Once the interpreter has run, it returns control back to the dispatcher loop in which eventually it will resume normal emulation.

5.4. Interrupt Details: OUT_OF_CODE

The `OUT_OF_CODE` interrupt is used to link caches together, by emitting a jump through the jump table to the next adjacent Chip8 memory location. This interrupt gets triggered whenever there is no more code to be executed, as a way to prevent the x86 `EIP` register from going out of the allocated memory region.

So far, this can only happen whenever a conditional jump is true, and it skips over the end jump within the cache. In this case, the next adjacent cache will contain the code to execute after the conditional jump. In order to reach this next cache, this interrupt handler will emit both a `PREPARE_FOR_JUMP` interrupt, and a `JMP PTR32` instruction through the jump table, eventually jumping to the cache end Chip8 PC plus 2 bytes.

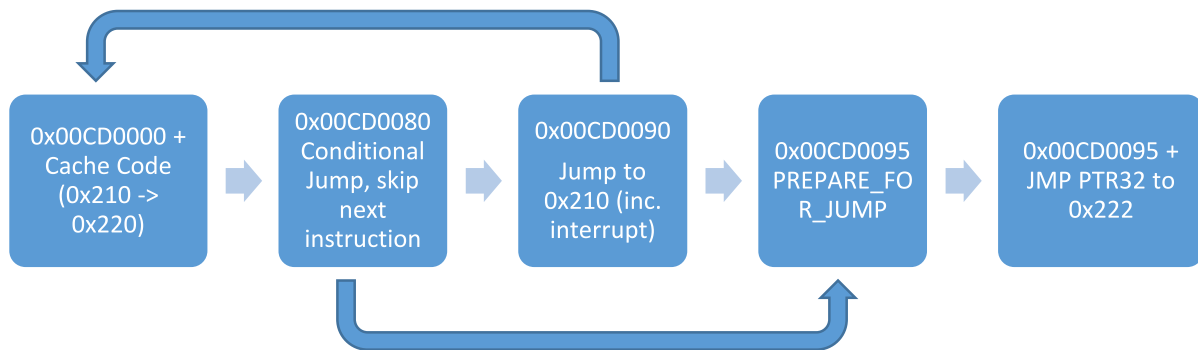

Figure 14: End result of the `OUT_OF_CODE` interrupt which adds on the extra jump at the end.

The only parameter parsed to this interrupt is the cache base address, which is known at the time when the interrupt is inserted into the end of the cache. Using this parameter, the handler is able to find which cache that corresponds to, and get the end Chip8 PC.

5.4.1. Differences from Other Interrupts

Earlier on in this document, I mentioned that the `OUT_OF_CODE` interrupt is not generated through the emitter, but is created at the end of a new cache as a way to protect the x86 instruction pointer from going out of bounds. In the case where the `OUT_OF_CODE` interrupt is reached, the handler is invoked from the dispatcher loop, just like any other normal interrupt.

Special care, however, must be taken to ensure that the `x86_resume_emulation` variable is reset to just after the last emitted instruction in the cache, before emulation is resumed. As this interrupt is raised at the end of the cache and not when it starts to run out of code, it is not possible to set the variable within the interrupt code. Luckily, it is easy to figure out where the `x86_resume_emulation` should point to, and this is the cache base address plus the `x86_pc` cache variable.

Note that in order to place this interrupt at the end when the cache is created, a cache size must be known. I have set this to be 4 KB of memory per cache, which is more than enough for any Chip8 rom.

6. Current Issues

6.1. Self Modifying Code

Currently self modifying code is handled by marking the caches that represent the memory written as invalid, meaning it should be deleted as soon as possible. However, currently this deletion only happens after two things:

1. There is a `PREPARE_FOR_JUMP` type of interrupt being serviced.
2. The `x86_resume_address` is not currently pointing to the marked cache.

An issue arises from this implementation, whereby if the memory written is actually in the same cache, and after the `SELF_MODIFYING_CODE` interrupt code in the cache, then the new code will never get a chance to run until the cache is deleted and recreated again.

So far I have not encountered a rom which requires the proper handling of this, so it may not actually be a problem and instead more of a general accuracy problem. I have not bothered to investigate a way to solve this, but a pull request is welcome!

7. Conclusion

After reading this document and looking at the accompanying source code, you should now have a grasp of how to make a dynamic recompiling emulator. There is a lot of work involved, and I'm sure some of you reading this may have better ways to do things. I created this emulator by myself as a way to learn about this area of emulation, so there are bound to be some areas for improvement.

Some of the significant concepts used in my implementation include the use of a dispatcher loop, interrupts, a jump table, and most importantly the core caches, emitter and translator. Without any one of these elements, the emulator will not work. Hopefully I have provided an adequate explanation of them.

If anyone reading this is still stuck, or you have some (constructive!) comments you may contact me through my email address listed at the beginning, or through various forums. Remember to consult Section vii for details about the files.

Good Luck!