**Introduction to Dynamic Recompilation in Emulation**  
Core Ideas + Chip 8 Example Using the x86-32 Instruction Set

Written by:

Marco Satti ([marcosatti@gmail.com](mailto:marcosatti@gmail.com))

Revision A

Contents

[i. Licence 3](#_Toc439180193)

[ii. Terminology 4](#_Toc439180194)

[iii. Helpful Resources 5](#_Toc439180195)

[iv. Prerequisites 6](#_Toc439180196)

[v. Purpose, Aim & Scope 8](#_Toc439180197)

[1. Introduction 9](#_Toc439180198)

[2. Core Concepts 10](#_Toc439180199)

[2.1. Caches 10](#_Toc439180200)

[2.2. Emitter 10](#_Toc439180201)

[2.3. Translator 10](#_Toc439180202)

[2.4. Relationship Between Caches, Emitter and Translator 11](#_Toc439180203)

[2.5. Interrupts 11](#_Toc439180204)

[2.6. Dispatcher Loop 12](#_Toc439180205)

[3. Problems with Dynamic Recompilation & Solutions (within Chip8) 13](#_Toc439180206)

[3.1. Jumps 13](#_Toc439180207)

[3.1.1. Jumps Problem 13](#_Toc439180208)

[3.1.2. Jumps Solution – Part 1 14](#_Toc439180209)

[3.1.3. Jumps Solution – Part 2 16](#_Toc439180210)

[3.2. Code Generation 17](#_Toc439180211)

[3.2.1. Code Generation Problem 17](#_Toc439180212)

[3.2.2. Code Generation Solution 18](#_Toc439180213)

[4. Dynamic Recompiler – Main Structure 19](#_Toc439180214)

[4.1. Interpreter Structure Review 19](#_Toc439180215)

[4.2. Dynamic Recompiler Structure Overview 20](#_Toc439180216)

[4.3. Dynamic Recompiler Structure – Part 1: Running Native Code 21](#_Toc439180217)

# 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/>

# 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, based on when the code is needed. |
| 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 instruction (that permits) is followed by the destination and followed by the source.

# 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) |

# Prerequisites

Some requisites listed here may or may not be useful to every aspect of this document. I have tried to categorise them in case you don’t want to follow the document exactly, but in general they are all going to be useful.

* **(Core) 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>

* **(Core) 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.

* **(Example) 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).

* **(Core/Example) 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>

* **(Core) How 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.

* **(Core/Example) Build an emulator using the interpreter method.**

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 instead of interpretation. This doesn’t necessarily have to be for the Chip8 either, any target architecture will do fine, since it all follows the same basic principles.

* **(Core/Example)** **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 & OpenGL, 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.

# 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:

1. **This document DOES NOT COVER EVERYTHING! There are MANY different ways to implement a dynamic recompiler – this is just one of them. If you are building a dynamic recompiler for another system, please go and do some research and/or develop your own ideas on how to solve problems. Make sure to go and look at similar architecture emulators for some inspiration if you are stuck! Usually there will be similar problems for the different architectures eg: ARM, MIPS, etc. I have built my emulator to understand dynamic recompilation and I am simply sharing what I know – it is not foolproof!**

# 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 program 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.

# 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.

## Caches

Caches are at the heart of a dynamic recompiler, where they are used to read and write 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 (see next Core Concept 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 3. This is why multiple caches are often used in emulators.

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.

Note: I use multiple caches in my emulator instead of just 1 for reasons mentioned later in Section 3.

## 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, some 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, you would have to be absolutely **mad** 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 it without this class (at least for me).

## 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.

Later on there will be references to a translator loop – these are not the same things! This section simply describes the core translator. The translator loop contains other logic, such as cache selection.

## 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 ii):

<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: Diagram of the relationship between the translator, emitter and caches.

Note: this is a top to 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 discussed in Section X.

## 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 (see Section 2.6, also known as the main program 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 the translated code which will cause the program to transfer control outside to the main program loop. The interrupt is then serviced by the appropriate handler, and transfers control back to the translated code.

More information will be provided in Section 4 where the dynamic recompiler structure will be discussed. Interrupts are a fairly simple concept, so there is not much else to explain here – just remember that they are used in the emulator but also why and how they work.

## 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 for the dispatcher loop will be only for point 2 – implementing the user interaction code will be up to you (I suggest looking at SDL if you are interested).

## 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:

Jump Table

Map Entry 1

Jump to Chip8 PC:

0x232

Cache x86 Address containing 0x232 start Chip8 PC:

0x0000B0000

Map Entry 2

Jump to Chip8 PC:

0x256

Cache x86 Address containing 0x256 start Chip8 PC:

0x0000D0000

(etc)

# Problems with Dynamic Recompilation & Solutions (within Chip8)

Before diving into the complete solution 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.

## Jumps

### 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 figure out the address it points to. 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 2: 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. 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):

**But where is 0x0200?**

**But where is the end of the next instruction?**

Figure 3: x86 example of jumps.

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 0x0NNN.

**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, GPU etc 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 0x0NNN + 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.

It is also interesting to note that this is largely where dynamic recompilation gets its name – if we didn’t need to do any runtime handling due to dependencies (the majority of which is jumps), we could actually just use another emulation technique called static recompilation. This is where we do not need to translate code or check jump locations during runtime, and instead would just need to do a single translator pass on the code which would be valid forever!

### 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.

Originally I had thought to just record the x86 address of EVERY Chip8 address that was translated – but quickly realised this is probably a bad practice when thinking about making dynamic recompilers for other target systems. One benefit of this method is that you would only need 1 cache to be created with a size large enough to hold all the code, making for very simple cache handling code. However, the drawback of this is that you will waste a lot of memory by storing address maps where roughly 90% of them would never be read from. I think this is a bad programming practice simply because of the relatively huge amount of wasted memory, but of course this method should be perfectly valid in order to solve the jumps problem. For example, if you were to create a dynamic recompiler for a 32-bit target system with megabytes of code to translate, you would be looking at a very large memory footprint. There may also probably be problems with self-modifying code, although I have not explored it.

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.

The advantage of this approach is that the memory footprint is reduced as now only the Chip8 addresses that are jump locations are stored in memory. Of course the disadvantage is that it makes the code quite a bit more complex – you are now responsible for choosing the cache that translated code should go into. In terms of scalability however, I think this is the better choice.

At this point, this is an example of what the cache layout would look like if we stopped here to take 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).
* Remembering that Chip8 opcodes take up 2 bytes of space.
* The caches have been allocated in x86 with sufficient space to hold the translated code and starts @ 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 are all known).

Figure X: Example of a cache layout.

Ok great – we now have a basis to deal with jump locations. But there are now problems introduced when running through the translator loop:

1. How will we know if the cache end PC is actually a jump instruction?
2. What if we encounter a jump location that is not to the start Chip8 address of a cache, but in the middle? If this is the case, we still do not know the x86 address of the Chip8 instruction (this happens frequently, especially when a loop is entered).

These two problems are solved by the use of flags used on each cache that are set and handled by the translator loop (respectively):

* stop\_write flag: this is the flag to control if a cache has a jump instruction written at the end, and should not be written to further by the translator loop. In other emulators, normally they would recompile code to a jump in one pass (ie: referred to a ‘block’ of code). In my design I had originally decided to translate 16 opcodes at a time for debugging. I have stuck with this decision although it is not necessary for it to work.
* invalid flag: this is the flag to control if a cache should be split up and the original deleted. Split up means to re-translate all of the code using the split regions.

More on both these flags in Section X about the translator structure, where it includes cache selection logic.

This leads us into Part 2, where we discuss conditional jumps. The main points to get out of Part 1 is:

* We are using multiple caches to deal with jumps, with cache parameters set on each one:
  + Memory Address (@): where the cache is allocated in x86 memory.
  + Start Chip8 PC: indicates where translated code starts for the cache. Must be a jump location.
  + End Chip8 PC: indicates where translated code ends for the cache. Must be a jump instruction.
  + stop\_write flag: indicates if the cache should be written to or not (ends on a branch).
  + invalid flag: indicates if the cache should be deleted as it has been split up.
* A cache is a continuous block of instructions that do not branch (there are no jump instructions anywhere within the cache until the end).
* From the above point, a cache should always finish on a x86 jump instruction after we have finished translating code for the block (otherwise the code will overrun).

The specific implementation of each of these three types of jumps are done through the translator loop and in interrupts - PREPARE\_FOR\_JUMP, PREPARE\_FOR\_STACK\_JUMP and PREPARE\_FOR\_INDIRECT\_JUMP. Only PREPARE\_FOR\_JUMP will be looked at later on, for the other two consult the source code.

### 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 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 relative jump 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, making sure it continues based on the condition that there is a conditional jump relative value to be filled in.
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.

Hopefully you’ll remember that caches should always end on a normal x86 jump instruction from Part 1. A problem that arises from this conditional jump solution is that often there are loop constructs that look like this:

Figure 2: x86 example of a loop construct.

As you can see, the conditional jump instruction will have nowhere to go to if it is true, even though the code is valid. This is because a different cache will contain the code that goes after the end instruction.

To solve this, we need to emit a jump to another cache, which contains the code after the cache end jump instruction. This is done by manually emitting an PREPARE\_FOR\_JUMP interrupt followed by emitting a x86 jump instruction into another cache. We can do this because an OUT\_OF\_CODE interrupt will be raised when the conditional jump is true and it skips over the end. Read further on to know why the OUT\_OF\_CODE interrupt will be raised.

## Code Generation

### Code Generation Problem

A problem arises from the cache structure changes made to the emulator in the solution from the jumps problem. From the previous problem, we are now dealing with caches in blocks of code instead of the whole program. There are 2 issues with this:

1. As we are dealing with 16 code translations at a time (instead of compiling whole blocks of code at a time), we need a way to determine if we should generate more code to the same cache after the translated code has run.
2. We need a way to determine if there has been a jump over the end of the cache (ie: a conditional jump has caused this), and if so, we need to jump to the correct cache.

Currently there is no safe guard in place at the end of caches if all of the translated code thus far is executed and no jump out of the cache occurs:

Case 1:

OR

Case 2:

Figure X: End of cache problem.

### Code Generation Solution

There needs to be a safe guard at the end of the cache in order for the x86 PC not to be out of the memory region allocated. This is in the form of an OUT\_OF\_CODE interrupt that is generated when the end of the cache is reached, which is placed there at the creation of the cache.

So far, we have defined some parameters for the caches created to deal with the jumps problem. One other parameter we need in order to develop a solution is to know how long the cache is (allocation length), so we can determine where to put this interrupt. In my emulator I have given each cache 4K of memory to work with, however this value could be larger or smaller depending on the game (I have determined 4K to be enough for any game).

One other requirement is for the cache at creation to be filled with **NOP**, so that it doesn’t change anything while reaching the end of the cache. I would imagine the speed penalty from doing this would be negligible, but I have not investigated it.

# Dynamic Recompiler – Main Structure

## 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 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 3: Interpreter Structure Overview.

In terms of source code, my Chip8 Interpreter example contains the following (modified):

|  |
| --- |
| // Emulation loop  **for** **(;;)** **{**  // Emulate one cycle  mChip8**->**emulateCycle**();**  // Handle Graphics, Sound, Input, etc  handleGraphics**();**  handleSound**();**  handleInput**();**  **}** |

Where mChip8**->**emulateCycle**()** looks like the following:

|  |
| --- |
| mChip8**::**emulateCycle**()** **{**  // Fetch opcode  opcode **=** memory**[**cpu**.**pc**]** **<<** 8 **|** memory**[**cpu**.**pc **+** 1**];**    // Decode opcode & handle (changes PC in the process)  **switch** **(**opcode **&** 0xF000**)** **{**  **case** 0x0000**:**  handleOpcodeMSN\_0**();**  **break;**  **.**  **.**  **.**  **case** 0xF000**:**  handleOpcodeMSN\_F**();**  **break;**  **}**    // Update timers  timers**->**updateTimers**();**  **}** |

## Dynamic Recompiler Structure Overview

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

Figure 4: 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.

Much like the interpreter’s translator loop, the dispatcher loop code is not terribly big:

|  |
| --- |
| // Emulator Program loop  **for** **(;;)** **{**  // Emulate one loop  mChip8**->**emulateLoop**();**  // Handle Graphics, Sound, Input, etc  handleGraphics**();**  handleSound**();**  handleInput**();**  **}**  mChip8**::**emulateLoop**()** **{**  // The heart and soul of this emulator  // Exec cache (first run will produce OUT\_OF\_CODE interrupt)  cache**->**execCache\_CDECL**();**  // Handle Interrupts  handleInterrupt**();**  **}** |

Note that this setup is not how a real emulator would be implemented – 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. Things like graphics, sound and input may need to be on separate threads for real-time handling. This serial-like structure works in the interpreter version as only one opcode is executed at a time, whereas blocks of code are executed at one time under a dynamic recompiler emulator which causes problems.

# Dynamic Recompiler Structure - Details

## 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 variable uint8\_t **\*** Chip8Globals::X86\_STATE::x86\_resume\_addressis introduced – this pointer variable stores the address at which emulation will resume.

Whenever an interrupt is generated through void Chip8Engine\_CodeEmitter\_x86**::**DYNAREC\_EMIT\_INTERRUPT(*params*) 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 previously stopped.

With this in place, we are now able to start and stop emulation at any time. In order to resume emulation, we need to transfer program control to the generated code, which is done according to the CDECL calling convention. Although the stack is not modified during the emulation at any point (except for the initial function call and return), it is better to implement the proper CDECL calling convention to allow for changes later involving the stack. There are two functions shown below which are both related to the emulation entry address and the CDECL calling convention.

|  |
| --- |
| class Chip8Engine\_CacheHandler  **{**  public**:**  uint8\_t **\*** setup\_cache\_cdecl**;**  uint8\_t setup\_cache\_cdecl\_sz**;**  uint8\_t **\*** setup\_cache\_return\_jmp\_address**;**  uint8\_t **\*** setup\_cache\_eip\_hack**;**    void setupCache\_CDECL**();**  void execCache\_CDECL**();**  **}**  void Chip8Engine\_CacheHandler**::**setupCache\_CDECL**()**  **{**  // A small cache which is used to handle the CDECL call convention before passing off  to the main cache execution point.  // Also contains the x86 EIP hack used to get the current EIP address and store it in  the eax register.  **if** **(**setup\_cache\_cdecl **==** **NULL)** **{**  // Alloc cdecl setup cache for first time. Will not change after this.  uint8\_t bytes**[]** **=** **{**  // Below code is used to 1. start CDECL calling convention, 2. jump to  emulation resume point, then 3. CDECL cleanup (return point).  // 1.  0x55**,** //0x0 PUSH ebp  0x89**,** //0x1 (1) MOV ebp, esp  0b11100101**,** //0x2 (2, MODRM) MOV ebp, esp  // 2.  0xFF**,** //0x3 (1) JMP r/m32  0b00100101**,** //0x4 (2, MODRM) JMP r/m32  0x00**,** //0x5 (3, DISP32)  0x00**,** //0x6 (4, DISP32)  0x00**,** //0x7 (5, DISP32)  0x00**,** //0x8 (6, DISP32)  // 3.  0x5D**,** //0x9 POP ebp  0xC3**,** //0xA RET  // HACK: ASM BELOW USED TO GET EIP ADDRESS AND RETURN IN EAX. SEE  CodeEmitter\_x86->DYNAREC\_MOV\_EAX\_EIP.  0x58**,** //0xB POP eax  0x50**,** //0xC PUSH eax  0xC3 //0xD RET  **};**  setup\_cache\_cdecl\_sz **=** **sizeof(**bytes**)** **/** **sizeof(**bytes**[**0**]);**  // WIN32 specific.  **if** **((**setup\_cache\_cdecl **=** **(**uint8\_t **\*)**VirtualAlloc**(**0**,** setup\_cache\_cdecl\_sz**,**  MEM\_COMMIT**,** PAGE\_EXECUTE\_READWRITE**))** **==** **NULL)** exit**(**2**);**  // Copy above raw x86 code into executable memory page.  memcpy**(**setup\_cache\_cdecl**,** bytes**,** setup\_cache\_cdecl\_sz**);**    // Update variables needed throughout program.  setup\_cache\_return\_jmp\_address **=** **(**setup\_cache\_cdecl **+** 0x9**);**  setup\_cache\_eip\_hack **=** **(**setup\_cache\_cdecl **+** 0xB**);**    // Update cdecl cache with location of x86\_resume\_address variable (will jump to  address contained in x86\_resume\_address).  **\*(**uint32\_t **\*)(**setup\_cache\_cdecl **+** 0x5**)** **=** **(**uint32\_t**)&**  Chip8Globals**::**X86\_STATE**::**x86\_resume\_address**;**  **}**  **}**  void Chip8Engine\_CacheHandler**::**execCache\_CDECL**()**  **{**  // Old method, doesnt work with optimisations turned on (optimises to JMP instead of  CALL).  //void(\_\_cdecl \*exec)() = (void(\_\_cdecl \*)())setup\_cache\_cdecl;  //exec();  // New method, works with optimisations turned on. TODO: Look at why we cant direcly  place value of setup\_cdecl\_cache into eax and call.. seems to put 14h instead of  address.. probably something to do with stack.  // CDECL calling convention, but there are no variables to push onto stack/remove from  stack by changing esp.  uint32\_t call\_address **=** **(**uint32\_t**)&**setup\_cache\_cdecl**;**  \_\_asm **{**  mov eax**,** call\_address  call **[**eax**]**  **};**  **}** |

After the program control has reached the jump instruction located in the CDECL cache that was setup, the processor will begin executing translated code. When an interrupt needs to be serviced, it will jump back to marker 3 within the byte array setup above (setup\_cache\_return\_jmp\_address), thus returning control back to the dispatcher loop according to the CDECL calling convention (stack is not modified).

There is one other part of the CDECL cache I should explain from above, and that is about the EIP “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 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.

## Handling Interrupts

This section will go over the different interrupts that will be encountered while running the translated code. 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

### 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, a JMP x86 instruction will be performed after this interrupt is serviced.

One may think that because the cache’s created already have a start Chip8 PC attribute, the x86 JMP instruction can just be a static jump to a specific address and therefore no preparing is needed. However, there are two problems with this:

1. We do not know x86 addresses ahead of time – that is we require the caches to be defined (this includes allocating the space in memory) before the start x86 address of the caches are known. For example, if we encounter a jump to 0x290 but we have only translated code up until 0x210, there is no way to determine ahead of time what the x86 address of the cache (starting with 0x290) will be.
2. If the jump is not to a starting Chip8 address, but into the middle of the cache, then we still have no idea what the x86 address should be. In this case the cache should be split up (and invalidated) into two smaller Chip8 PC subsets.

To resolve this, a jump table is created, which involves holding all of the jump locations in a table, where each entry is mapped to a x86 address value that can be dynamically updated in the PREPARE\_FOR\_JUMP interrupt. When a JMP instruction is emitted, it points to the x86 address variable in the jump table which contains the correct jump location.

Cache 1

S: 0x210

E: 0x230

0x0000A000

Start of subroutine

0x0000A090

End of subroutine

0x0000A094

Jump to next cache (0x232)

JMP PTR32

Jump Table

Map Entry 1

Jump to Chip8 PC:

0x232

Cache x86 Address containing 0x232 start Chip8 PC:

0x0000B0000

Cache 2

S: 0x232

E: 0x250

0x0000B000

Start of subroutine

0x0000B004 +

(more code)

As a cache that contains the jump location Chip8 PC may not exist when a jump is performed, the PREPARE\_FOR\_JUMP interrupt makes sure these jumps have somewhere to go. It does this by firstly iterating through all of the jump table entries, making sure all of the entries have an x86 address to jump to. If there are any entries that are not valid, an empty cache will be created corresponding for the jump Chip8 PC. Before performing this check, all caches that have the invalid flag set will be deleted, sparing a cache if x86\_resume\_address is currently within that cache (it will be deleted once it is no longer used).

An empty cache is created using the same flow logic presented below in the OUT\_OF\_CODE interrupt.

### 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. This signals the dispatcher loop to use the interpreter to handle the opcode.

Dispatcher Loop

Start

Run Caches

Handle Interrupt

Finish

(etc)

Caches

< 0x0000A000

Previous x86 code

0x0000A030

DXY0 opcode (USE\_INTERPRETER)

Interpreter

Decode opcode given

Perform DXY0 (draw opcode)

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

### Interrupt Details: OUT\_OF\_CODE

Earlier on in this document, I mentioned that the OUT\_OF\_CODE is not generated through the emitter, but is created inside a new cache at the end, 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. There are 3 cases where the OUT\_OF\_CODE interrupt will be reached:

1. The cache is genuinely out of code (as only 16 opcodes are compiled at a time) and needs more code until a jump is encountered.
2. There is a conditional jump, where it has gone over the next cache instruction (jump) as the underlying condition is true.
3. A cache has previously been split up into two, due to a jump being needed into the middle of the cache. This means although there are two separate caches, no jump will ever exist in the first cache due to no jump being in the Chip8 rom.

Case 1 is a unique case, where the solution to this problem is to translate more code until a jump is encountered. Case 2 & 3 are similar cases, where they both require a jump into the next cache of Chip8 End PC + 2 (eg: jump needed for 0x210 -> 0x212).

In order to distinguish between the two unique code paths, we can check if the stop\_write flag is set, and also check if there is an existing cache which has the next Chip8 PC as the start PC. In summary, this is a table of the possibilities based on these two parameters:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Case 1 | Case 2 | Case 3 |
| stop\_write flag | 0 | 1 | 0 |
| Existing cache for End PC + 2 | No | Yes or No | Yes |

Based on these two parameters, we can determine if the translator loop needs to run or if a jump needs to be emitted into another cache. To simplify the code path logic, only Case 1 is checked for in an if-else statement, where the code path for Case 2 and 3 is within the else statement. With the above parameters, all of the possibilities have been accounted for.

Sometimes a problem occurs with the code path selection after a cache has been split up, due to a jump into the middle of the original cache. If either of the split cache Chip8 ranges are only 1 opcode long and it is run for the first time, it will incorrectly be recognised as Case 3 instead of Case 1. To get around this as a workaround (this will need a better fix), within the if statement, the amount of translated code is checked for as being equal to 0 in a logical OR arrangement with the above conditions. If this condition is true, it will follow the Case 1 code path. In practice, the implementation code is displayed below.

Note that before the interrupt is handled, any caches that are marked invalid are deleted.

|  |
| --- |
| void Chip8Engine**::**handleInterrupt\_OUT\_OF\_CODE**()**  **{**  // ! ! ! X86\_STATE::x86\_resume\_c8\_pc contains start pc of cache, X86\_STATE::x86\_resume\_start\_address contains starting x86 address of cache ! ! !  // Flush caches that are marked  cache**->**invalidateCacheByFlag**();**  // End of cache was reached, because of 3 posibilities:  // 1. The cache is genuinely out of code due to eg: translator loop exiting on X many cycles  // 2. The cache contains code (ie: conditional jump) that goes over the end jump  // 3. The cache has previously been split up into two caches due to a jump into the middle of the cache (and needs to be "joined up").  // We can determine which case it is by looking at the do not write flag, where 1 and 3. will have false, and 2. will have true.  // For determining between 1 and 3, we can check for an existing cache. If true, then it is case 3.  // However, case 2 and 3 can be solved in the same way, by emitting a jump to (C8 end PC + 2).  // In summary:  // Case 1: stop\_flag = 0, existing cache = 0  // Case 2: stop\_flag = 1, existing cache = 1 or 0  // Case 3: stop\_flag = 0, existing cache = 1  // Together, Case 2 & 3 make up the complement possiblilies to Case 1 (ie: IF (Case 1) and ELSE (Case 2 & 3) statement will work).  // PROBLEM:  // There is sometimes a problem, where there has been a cache that has been split up where both of them are only 1 Chip8 opcode apart, eg: 0x206 and 0x208  // In this case, the cache with start C8 pc = 0x206 will also have an end C8 pc = 0x206 but does not contain any translated code for the 0x206 PC.  // This means that on the first OUT\_OF\_CODE interrupt from this cache, it will be detected as stop\_flag = 0 and existing cache = 1 (Case 3), but actually it should be Case 1 as the code has not been translated yet.  // After the code has been translated, thats when its safe to detect as Case 3.  // The solution is to check within Case 1 if also no translated code exists by x86\_pc == 0 (logical OR). Has not been thoroughly tested and may present problems in other games.    // There is possibly a better way to do this...  #ifdef USE\_DEBUG  cache**->**DEBUG\_printCacheList**();**  printf**(**"----\n\n"**);**  #endif  // Get cache details that caused interrupt.  int32\_t cache\_index **=** cache**->**findCacheIndexByX86Address**(**X86\_STATE**::**x86\_interrupt\_x86\_param1**);** // param1 should be the base address of the cache, so we can find the cache that interrupted by searching for this value.  CACHE\_REGION **\*** region **=** cache**->**getCacheInfoByIndex**(**cache\_index**);**  // Remember to reset the entry point to the current cache PC.  X86\_STATE**::**x86\_resume\_address **=** region**->**x86\_mem\_address **+** region**->**x86\_pc**;**  // Case logic  // Case 1  **if** **(**region**->**x86\_pc **==** 0  **||** **(**region**->**stop\_write\_flag **==** 0  **&&** cache**->**findCacheIndexByStartC8PC**(**region**->**c8\_end\_recompile\_pc **+** 2**)** **==** **-**1**))** **{**  // Start recompiling code at end of cache c8pc  C8\_STATE**::**cpu**.**pc **=** region**->**c8\_end\_recompile\_pc**;**  translatorLoop**();**  **}**  // Case 2 & 3  **else** **{**  // First get jump table entry  int32\_t tblindex **=** jumptbl**->**getJumpIndexByC8PC**(**region**->**c8\_end\_recompile\_pc **+** 2**);**  // Emit the jump  int32\_t old\_index **=** cache**->**findCacheIndexCurrent**();**  cache**->**switchCacheByIndex**(**cache\_index**);**  emitter**->**DYNAREC\_EMIT\_INTERRUPT**(**X86\_STATE**::**PREPARE\_FOR\_JUMP**,** region**->**c8\_end\_recompile\_pc **+** 2**);**  emitter**->**JMP\_M\_PTR\_32**((**uint32\_t**\*)&**jumptbl**->**getJumpInfoByIndex**(**tblindex**)->**x86\_address\_to**);**  cache**->**setStopWriteFlagCurrent**();**  cache**->**switchCacheByIndex**(**old\_index**);**  **}**  **}** |

## Translator Loop Details