ROM Hacking 202

Finding and reversing the decompression algorithm

You will need:

- BGB emulator
- The example game's ROM: Kikansha Thomas - Sodor-tou no Nakamatachi (J) [C][!]
- Some comfortability with programming concepts
- `Determination`
- Python 3, if you want to run the tool; but it also makes for a good calculator

If you have not yet done so, please read the ROM Hacking 201 document by abridgewater.

Throughout this tutorial I try to use the format for addresses and values you will see on screen. These can differ in that some will be a prefix plus a hexadecimal address like ROM0:1084 and some will simply be a hex address such as 1084. When discussing abstract hexadecimal values, I'll use the 0x prefix as in 0x1084 and they will be ordered as you type them into a calculator. Regardless of the format, all numbers in this document (besides the title, of course) are hexadecimal (or under 10 so it doesn't matter).

Preface

Following off an early version of abridgewater's work on the decompression algorithm, I set out to implement a tool to decompress the graphics. The algorithm was not *quite* complete so I had to go looking at the game code to figure it out. Thus, this will be a tutorial on how to reverse engineer the algorithm, in part, from the assembly using the BGB debugger.

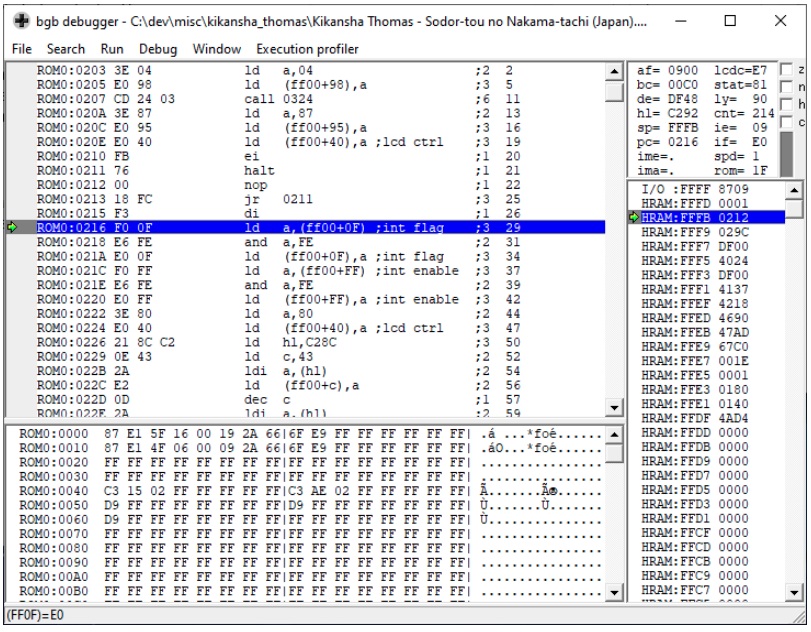
I had no background on Game Boy itself before starting this endeavor, and as such, this will be written from that perspective. I did, however, have a good deal of ASM experience from other situations and will do my best to explain what of that experience was useful in reversing this algorithm.

To start, a summary of the algorithm as described by abridgewater:

> There is a 4 byte header, the first two-byte value is some kind of offset (read: unknown use), the second is the size of the compressed data. The compression algorithm has a control byte which is read LSB to MSB (0bE000000S, S = start, E = end). If the bit is a 1, copy the next byte to the decompression stream as-is. If the bit is a 0, the next byte is the relative offset indexing backwards (from the cursor) into the decompression stream such that a value of 0 is the most recent byte, a value of 1 is the byte before it, etc. The following byte is the amount to copy.

In terms of background assembly information, it's useful to be familiar with common mnemonics (aka operations) such as JR (Jump Relative). These will be discussed as they come up, because there are quite a few. Before that, it's important to be familiar with the concepts of how a processor works at a high level, and what all the parts are that you will see on the debugger screen and other tools.

If you haven't already, set up BGB and load up your Kikansha Thomas ROM. Right-click the screen and select "Other → Debugger" to open the debugger. You'll see a window like this:



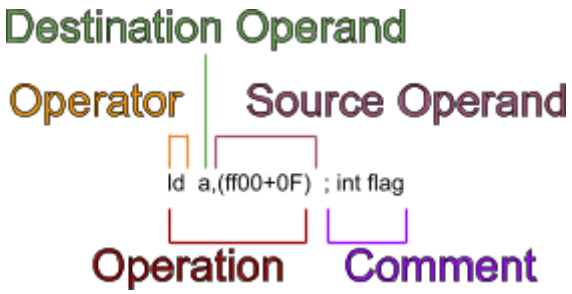
There are four boxes on this screen. The upper-left is the disassembly view, the upper-right is the register view, the lower-left is the memory view, and the lower-right is the stack view. Now for what these terms (and more!) actually mean:

Q Disassembly

A This is the interpretation of the bytes as code. Reading from left to right in the disassembly view, you see: the address of the bytes, the bytes themselves, the Z80 disassembly itself, and then some comments (anything following a semicolon). Just because something disassembles does not mean it's code! Data is arbitrary! It's *only* code if it's executed.

The blue line in this view in the screenshot above is where the Program Cursor currently is. It means that, when the execution is resumed, that operation will be the next one to execute. Thus, it *is* code.

Of the disassembly, there are a few terms to understand:



The operation (aka instruction) is all the parts: the operator and its operands. It can also refer to the bytecode it assembles to. This is enough to perform some simple instruction which changes the state (the registers, I/O, memory, etc) for instance a single addition. If an operation changes the value in either operand, it will always be the first one, hence calling it the destination. However, in BGB syntax, for many arithmetic operations, the only destination can be A, so it's elided. That is, if you see the operation `add b` then this is equivalent to the C code `A += B` or `A = A + B`. That is, "Add A and B then put the sum in A".
The operator (aka mnemonic) is essentially the name of the function you want to perform. It's a shorthand name because this was popular in the last century due to the math influence. You can look up a list, but you only need to if you're interested. The ones we use will be explained during the tutorial.
The operands are arguments to the operation. They are not *everything* the operation may affect, however. Most operations will also affect the flags and sometimes affecting the lower half of a composite register will affect the upper half (more on registers below) or vise-versa.
Comments are informative and are not part of the instruction. They're prefixed by a semicolon (;), as discussed above.

Q Registers

A Registers are essentially variables that live inside the CPU. Some of them have a special meaning and some don't. They have a certain size, measured in bits, and some of them can be decomposed. For instance, `DE`'s bytes can be addressed as `D` and `E` separately.

For our purposes you will need to know that all CPUs have a Program Counter (usually called `PC`, as it is here) and a Stack Pointer (usually called `SP`, as it is here). There is also a flags register, whose name always changes, but you can see the checkboxes on the right-hand side here which are the individual flags.

The flags register is a bitfield, meaning its individual bits represent binary states, hence why it's represented by checkboxes. There are two flags you'll find on every system which you need to know: `Z` (for Zero) and `C` (for Carry). For many arithmetic operations, if the resulting value is zero, the `Z` flag is set. Carry flag can differ slightly between systems, but it always has to do with an overflow of some kind; for Game Boy it's an unsigned overflow (so passing the 0xff/0x00 boundary in either direction). Importantly, the carry flag can also be used in certain arithmetic operations. For instance, if you find `RRC` (Rotate Right through Carry) you can assume the carry flag will be included in the rotation somehow (the exact details are unnecessary).

It's important to understand the concept of flags because you will find what are called "conditional jumps" or occasionally Jcc. In BGB these are disassembled as something like `jr nz,024C` which reads as "Jump Relative if Not Zero to the address 0x024C" (BGB calculates the absolute address for you and is showing that, in the bytecode it's a relative address, hence the name).

Q Memory

A The memory includes both the ROM and the RAM. In BGB, the sections are prefixed by the name of the memory so it's easy to identify. ROM0 is bank 0 of the ROM, RO## is bank ## of the ROM, VRA# is bank # of the VRAM, and so forth. You won't need to memorize these or know them in too much detail. The other tools BGB provides will help us to understand these in the way that we need to.

From this view we can search the memory and set access breakpoints, which will be important later.

Q Stack and heap

A The stack is a section of RAM (in GBC's case, HRAM) which is used to store information temporarily. Registers will be pushed to the stack occasionally for

various reasons, and, importantly, return addresses (from `CALL`s) will be pushed to the stack and used by the `RET` operations to return execution flow to the point of origin.

Pushing some value to the stack means to add that value to the top of the stack (and adjust the stack pointer so that it points to that new location). Popping a value means to remove that value from the stack and load it into some register (which will be specified) then adjusting the stack pointer to point to the address of what was under that value in the stack. If you think of this like a stack of dishes, pushing a value to the stack is adding a new plate on top, and popping is removing the top-most plate. The top of the stack is literally the top-most plate currently on the stack.

The stack view looks like an upside-down version of the memory view because it is. The stack is nearly always located at the end of available RAM and its contents can be accessed and manipulated in the same way as normal memory. The head of the stack is pointed to by SP; and when something is pushed to the stack, SP is decremented by that size, because the stack grows upwards from the bottom of RAM (like a stack of dirty clothes). The heap is the unused portion of the potential stack space, from the address at which it's safe to consider this memory space to be reserved for the stack (the start of HRAM, probably) until SP. When you compile C code, local variables will be allocated on the heap.

Q Breakpoints (BP)

A There are two kinds of breakpoints we care about. In BGB they're called Breakpoints and Access Breakpoints. Respectively, these are breakpoints for execution and for memory access. You can place the former by double-clicking a line in the disassembly view and the latter by right-clicking a byte in the memory view and selecting Set Access Breakpoint. The details of these will be discussed as they're used, but keep in mind that this essentially means you can stop execution whenever some memory (or code; remember: arbitrary) is executed, read from, or written to; and you can select which.

Q Tiles, sprites, etc

A In embedded systems like this, there tend to be graphics controllers which have simple ways to put graphics onto the screen. The programmer doesn't draw pixels to the screen, but instead instructs the renderer about where the tiles and sprites are located in memory and how to render them onto the screen.

Tiles are, for GB, 8x8 graphics which can be placed on the screen in order to make a background. They must be placed at 8 pixel boundaries, meaning they can't overlap.

There are two important concepts here, the tile*set* and the tile*map*. The tileset is the actual graphics data, the pixels themselves, which on GBC is a 2bpp (2 bits per pixel) sequence that indexes into a palette which is assigned separately, which you can see in the “Tiles” tab of the “VRAM Viewer”. The tilemap is the assignment of those tiles onto the background render, which you can see in the “BG map” tab of the “VRAM Viewer”.

We’re not dealing with sprites in this tutorial, but they’re controlled by a sprite set (or sprite sheet) and the OAM (Object Attribute Memory). They’re used to display graphics which can move over the background, like game characters.

Q Little endian

A Almost every platform you’ll encounter uses little endian, which means the least significant byte is listed first for multi-byte values in memory. This *does not* apply to registers or the disassembly view or anything like that, it only applies to raw memory.

As an example, if you see a byte sequence of 55 AA and you happen to know that it is indeed a 16-bit value, the real value will be 0xAA55.

For reference, big endian is the opposite and is also called network byte order. When people refer to the endianness of a system, they’re talking about whether it’s big or little endian.

Q References and dereferencing

A A reference (aka pointer) is an address which represents a location in memory and is stored as a regular number. Remember, data is arbitrary, everything is just numbers. So long as the number is within the valid memory range, it can be treated as an address. Treating a number as an address, and reading data from that location in memory, is called dereferencing. In BGB’s syntax this is represented as enclosing the register whose value is being used as a reference in parentheses. For example, `ld a,(hl)` if HL contains the value 0x1000, then you can, in the memory view, “right-click → Go to…” 1000 and click ok to be taken to that location in memory, and see what data will be loaded into A.

The real tutorial

What we need is the location of the decompression algorithm, in full. So, how? Well, we know what we're decompressing (graphics), and where those are in memory, sort of. So let's get the specifics. Make sure the game is on the title screen, as that's the graphics we'll be working with. Now pause the game if it's not already. We go to Window → VRAM Viewer from the debugger window to open the graphics viewer, and we want to look at the Tiles tab because the title screen is made up of tiles. We want to pick some distinct tile, so one that's not a solid color, for instance tile 03 in the lower left pane. It shows an address of 0:9030 so let's go take a look there in the memory view. Right-click in the memory view and select "Go to…" then type "0:9030" without quotes and click OK. It will take you to VRA0:9030 where you'll see the following:

```
VRA0:9030   00 FF 00 FF 00 FF 00 FF|00 FF 00 FF 38 C7 FE 01| ............8Çþ.
```

The 38 is unique enough for our purposes, so we'll use that one. We want a distinct one, because it's less likely to be written as that value for any other purpose besides the one we care about. So click the 38 and right-click and select "Set access breakpoint" which will pop up the screen with the "addr range" field pre-filled. Make sure "on write" is the only checkbox which is checked (the default), type 38 into the "value" field, and finally click the "Add" button. Now, in order to trigger this BP, since the graphics are already decompressed, we need to restart the game. Right-click the game screen and select "Reset gameboy *" then select "Run → Run" from the debugger window. The BP will immediately be triggered at this line:

```
ROM0:18DA 12                ld    (de),a                ; 2  26
```

This operation is "Load A into the address pointed to by DE". Essentially, as expected, it writes A (which contains 0x38) to the memory position we asked the emulator to watch. Note this address (tip: you can set a Breakpoint here then disable it from the "Debug → breakpoints" menu in order to save it as a bookmark). However, typically the function we end up in from doing this isn't the one we need, so we want to go higher. BGB has a nice function for this! Select "Run → Step out" to go up to the calling function. We will end up on the address *after* the call to this function, which will look like this:

```
  ROM0:1081 CD A4 18        call 18A4                   ; 6 19
➡ROM0:1084 CD 9C 10         call 109C                   ; 6 25
```

So the function we were in starts at ROM0:18A4, also an important thing to note. So where are we now? It might be what we need, but we need to know where it starts. If we step out again, we find it was a call to 1048. Great! Let's go there and set a BP. Right-click, select "Go to…" and type in "1048" without quotes then click OK. It will jump to the start of the function and we can double-click to set a BP there. Now restart again and we'll see what this does.

Stepping through this code is a bit long and complicated. The important thing to do is watch the registers and how they change as well as what memory is being accessed. But we don't have to

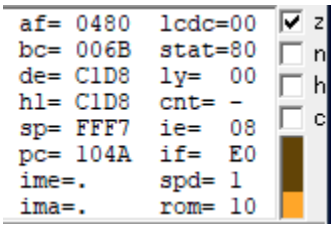
go one step at a time and decompile the thing, we just want to know some specific information. So let's look at the call we want and work backwards to see what's important, and what we should watch out for. Scroll back down to ROM0:1081 and we can see it sets values to H and L as well as pushing HL to the stack before doing so. Because there's a pop HL right after the second call, we can assume that the push HL is irrelevant to the call (it's just being stored because it's important to *this* function, and needs to be restored after these calls) so likely the arguments to 18A4 are simply H and L (or HL as a whole, since they're the decomposition of that). If we remember from earlier, DE contained the address of the destination in RAM, so it may also be important that D and E are assigned to just a bit earlier than H and L. Let's take a look at the supposed decompression function at 18A4 to see if DE has changed at all. A quick visual scan between 18A4 and 18DA shows that indeed, DE has no reassignments, so that load is important to us. We could go through and figure out every important thing, but let's see where we can get from here. We can always go back (tip: if you're working on something that's not at the start of the game, use save states).

```
ROM0:1048 F0 98           ld   a,(ff00+98)          ;3  6
ROM0:104A 5F              ld   e,a                  ;1  7
ROM0:104B B7              or   a                    ;1  8
ROM0:104C 28 3F           jr   z,108D               ;2  10
ROM0:104E 3D              dec  a                    ;1  11
ROM0:104F 28 41           jr   z,1092               ;2  13
ROM0:1051 3D              dec  a                    ;1  14
ROM0:1052 28 42           jr   z,1096               ;2  16
ROM0:1054 7B              ld   a,e                  ;1  17
ROM0:1055 C6 0B           add  a,0B                 ;2  19
ROM0:1057 87              add  a                    ;1  20
ROM0:1058 5F              ld   e,a                  ;1  21
ROM0:1059 16 00           ld   d,00                 ;2  23
ROM0:105B 21 C7 10        ld   hl,10C7              ;3  26
ROM0:105E 19              add  hl,de                ;2  28
ROM0:105F 2A              ldi  a,(hl)               ;2  30
ROM0:1060 66              ld   h,(hl)               ;2  32
ROM0:1061 6F              ld   l,a                  ;1  33
ROM0:1062 F0 9B           ld   a,(ff00+9B)          ;3  36
ROM0:1064 EA 00 20        ld   (2000),a             ;4  40
ROM0:1067 2A              ldi  a,(hl)               ;2  42
ROM0:1068 4F              ld   c,a                  ;1  43
ROM0:1069 2A              ldi  a,(hl)               ;2  45
ROM0:106A 47              ld   b,a                  ;1  46
ROM0:106B 2A              ldi  a,(hl)               ;2  48
ROM0:106C 5F              ld   e,a                  ;1  49
ROM0:106D A0              and  b                    ;1  50
ROM0:106E A1              and  c                    ;1  51
ROM0:106F 3C              inc  a                    ;1  52
ROM0:1070 C8              ret  z                    ;2  54
ROM0:1071 7B              ld   a,e                  ;1  55
ROM0:1072 F5              push af                   ;4  59
ROM0:1073 2A              ldi  a,(hl)               ;2  61
ROM0:1074 5F              ld   e,a                  ;1  62
ROM0:1075 2A              ldi  a,(hl)               ;2  64
ROM0:1076 57              ld   d,a                  ;1  65
ROM0:1077 2A              ldi  a,(hl)               ;2  67
ROM0:1078 E0 4F           ld   (ff00+4F),a ;vram bank ;3  70
ROM0:107A F1              pop  af                   ;3  73
ROM0:107B EA 00 20        ld   (2000),a             ;4  77
ROM0:107E E5              push hl                   ;4  81
ROM0:107F 69              ld   l,c                  ;1  82
ROM0:1080 60              ld   h,b                  ;1  83
ROM0:1081 CD A4 18        call 18A4                 ;6  89
ROM0:1084 CD 9C 10        call 109C                 ;6  95
ROM0:1087 E1              pop  hl                   ;3  98
```

The two important commands for stepping through the code are Step Over (hotkey: F3) and Trace (hotkey: F7) (normally called Step Into in other debuggers). Most of the time we'll use Step Over, but when we get to the call to ROM0:18A4, we'll use Trace. Remember, we're looking for what data HL and DE contain and where they come from. We know that DE contains the target address, but not where it comes from. We don't know anything about HL yet.

If you know any of the mnemonics in the above, you can work out a bit of what's about to happen. Let's trace backwards. HL is set from B and C (107F, 1080), DE is set from A (1074, 1076) which is set by the value at an address indicated by HL (1073, 1075), BC are also set by an address in HL (1068, 106A), HL is 10C7 (105B) + DE (105E), and that DE is has a high (D) of 0 (1059) and a low (E) of some longer arithmetic which originates with a value at 0xff98 on the heap (1048). So let's step through (F3/F7, as appropriate) and watch how A is calculated.

It first pulls the value from 0xff98 giving us this register view:



It may not be incredibly apparent since this is our first time looking at it, but A is the left two digits in that number, the 04. And we can also see that in the stack view, if you scroll down to HRAM:FF97 there's a value of 0400 where the 04 has the address FF98 and the 00 has the address FF97.
Next, it moves that value into e then performs a check. While OR means a bitwise or as one might expect, since it's against A, what it's actually doing is, in C syntax, `A | A` which means it's essentially not doing anything. The value of A doesn't change. However, the *flags* change. If A was 0, this would "set" the value of A to 0, causing the zero flag to be set. The next operation is a conditional jump saying that, if A was 0 (and this operation thus set the 0 flag), it should jump to 108D. Since A was 04, the Z flag was unset and it does not jump.
Next it decrements A, setting it to 03 and there's another Z check. If A had been 1, and thus became 0 after the DEC, it would jump to 1092. Then it decrements A again, making it 2, and has the same check again. Now it restores the original value back to A, from E.
Next it adds 0B, making A = 0F. Then it adds A to itself, effectively multiplying it by 2, which sets A to 1E. It puts that back in E and uses it as an offset from 10C7.

So the final math, in C syntax, for what just happened is this: `HL = 10C7 + (*(ff98) + 0B) * 2` with a final value of 10E5.

Now it actually starts reading from this address; so let's jump to it in the memory viewer and see what it is. It's located in ROM, so it's probably important to us, and we should write down the address for later.

```
ROM0:10E0   12 21 13 84 13 60 12 7B|13 9F 13 C0 13 F3 13 20| .!...`.{...À.ó.
```

So as we step through these assignments we can see A is set to 60 (and HL increments by 1, so that's what `LDI` does!) and H is set to 13. A then moves to L and it uses A to transfer a value from the heap at ff9B (01) to the memory location stored in 2000 (which is 0x208F ?? dunno where that goes off-hand, don't care). The important part is HL has changed again, to 1360.

```
ROM0:1360   00 40 13 00 90 00 38 45|13 00 90 01 B6 49 13 00| .@....8E....¶I..
```

This is read into A which is copied to C, then A which is copied to B, then A which is copied to E. So C = 00, B = 40, and E = 13. Now there's some bitwise and calls, an INC, and a RET Z. Hopefully you guess it, but this is another check. RET Z only returns if the zero flag is set, so it is checking to see if `(A & B & C) + 1 == 0`. Under what conditions would this be true? Only if A, B, and C were all 0xFF (because `0xFF + 1 == 0`… when working with 8-bit values).
So the check doesn't fire and it continues with this function. It puts E (13) back into A and pushes AF (this function hasn't assigned a value to F at all, but it's currently 0). Now it loads the next 3 bytes into E, D, and ff4F, which BGB helpfully informs us refers to the VRAM bank. So we can infer that these 3 bytes are probably a VRAM address, and 0x9000 does make sense for tile graphics (if you remember from our access BP, it was at 0x903C). This is definitely what we need! And it is assigned to DE, so it makes sense. Note this ROM address and put a nice sticker next to it.
That 13 from earlier is written to the 2000 indirection again, and then it loads BC into HL. BC was from those first bytes at 1360, with a value of 4000, so the 13 and thus this 2000 indirection are probably related, but who knows. Hopefully us, soon.

It's that time now, we need to trace into ROM0:18A4 using the F7 key. We can see it immediately increments HL twice then starts reading from it, so HL is clearly an address. Let's go there. This is the data from abridgewater's work! So we've found the compressed data, in bank 13 address 4000. Oh, we know that 13 don't we? So those three bytes were the banked address. What a win. The data at 0x1360 must be some sort of graphics table which terminates with 0xFFFFFF. We'll look into that later. It skips the first two bytes of the header, so that's not important for reading the compressed data. It reads the next two bytes and stores them in BC. Then it stores 0 in 0xff9E and 0xff9F. This immediate load at 0x18AF after an assignment of the same thing, indicates to me that this is the start of a loop (if this is the target of a later jump, it makes sense to do this, because the value of 0xff9F will have changed).

Then it starts performing math that needs to be stepped through for you to understand, and even then probably needs an explanation. SRL and RRA are types of shifts and rotations. With these mnemonics you can remember it like this: the first will be S or R to indicate a Shift or a Rotation. Shifts eliminate bits from one side whereas rotations wrap them around. Then somewhere in the acronym you'll see L vs A which means Logical vs Arithmetic. This matters for shifts when the value is negative (the MSB is 1) in that an arithmetic shift right will load a 1 into the MSB, but a logical shift right will load a 0. The distinction may or may not matter for other operations (left shifts or rotations) but theoretically it doesn't. Lastly the third element of the

acronym is either L or R for Left or Right. There may be a C somewhere near the end of the acronym (in GBC, like RRCA) which indicates that it goes through the carry flag as if it were an extra bit in the register. You will see these in action as you step through.

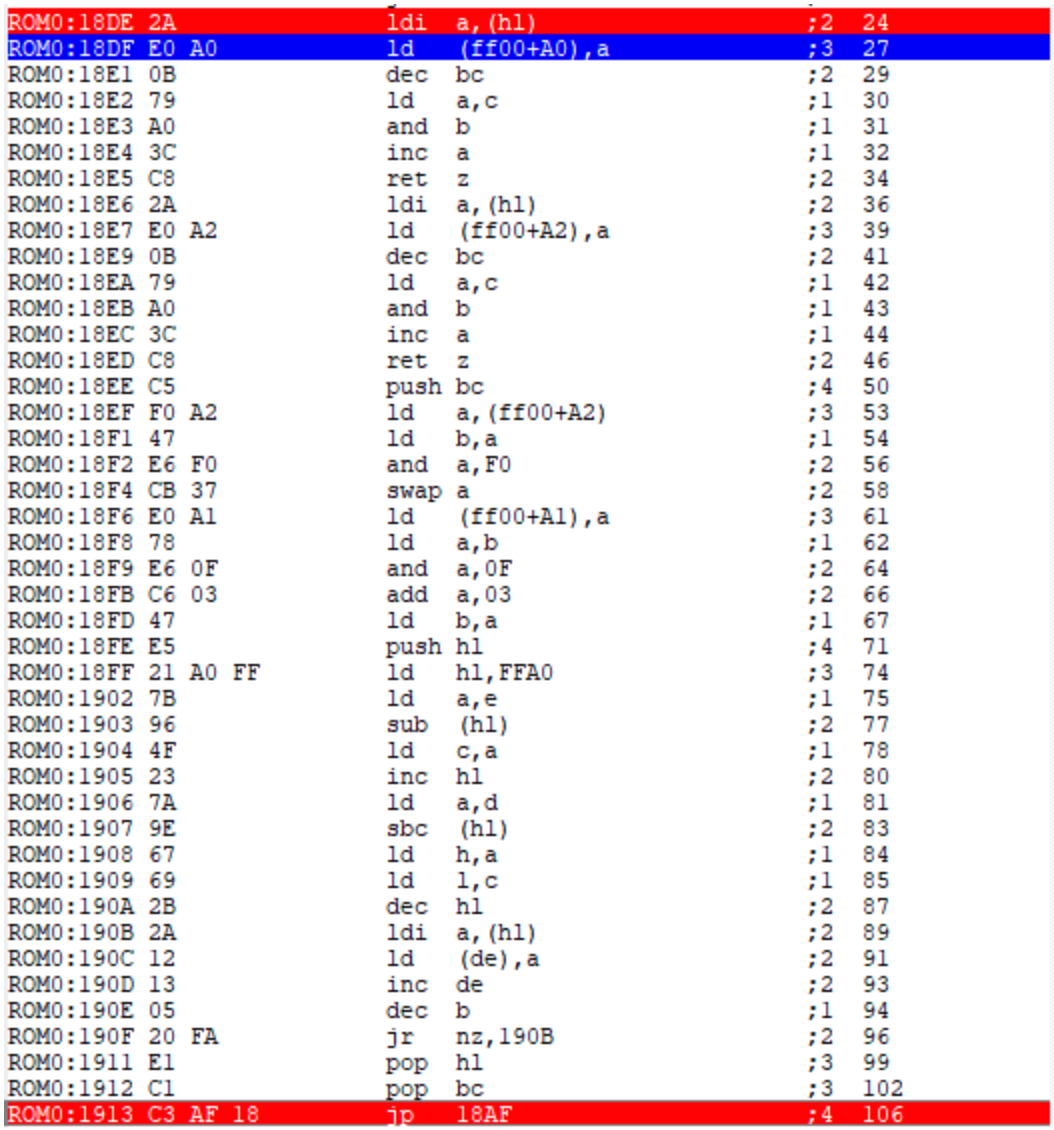
Now as we step through this algorithm, remember what we know: there's a control byte which specifies what the next 8 records are, if the corresponding bit is 1 then the record is a single-byte literal, if it's 0 then the record is a 2-byte repetition description. I will spoil you now and say that the original description does not describe the repetition description correctly, as I tested it by implementing it and attempted to decompress some graphics, so our goal here is to figure out how that sequence actually works. In order to do that, we need to find the part of the algorithm which uses those bytes.

You can find the section needed with an on-read access breakpoint, but since we want to understand the control flow better and it's probably not far, let's step through the function and see what happens. Step over ROM0:18BF and we can see the first control byte, 0x65, is loaded into A then 0xff9E. ROM0:18C3~ROM0:18CB (where it restores A from 0xff9E) seems to be some sort of check on BC, which stores the second 16-bit int from the header. We don't really care about this so let's skip to when A is restored. The next command is RRCA which, if you remember from above, is Rotate Right with Carry Arithmetically. So let's convert 0x65 to binary in order to see what to expect. A quick way to do this is to keep a Python console open and use the command `bin(0x65)` which returns 0b1100101. So the LSB is set, we can expect C to be set afterwards. The JR NC then decides where we go. If C is set we expect to go to the copy of a literal byte, which we don't care about. If it's unset, we expect to go to where we want. So, since this won't go where we want, we can set a BP on ROM0:18DE ourselves and press F9 (for Run).

We're now where we want to be. We need to decompile this code, so let's read it over and figure out what it means. If we glance through the code for branching operations (JR, JP, RET) we see conditional RETs, one conditional JR which jumps backwards, and finally an unconditional JP. So this JP is the absolute end of this (or the total) code section. We can then just look over most of this and decompile it. A screenshot of this section is below.

First we take an inventory of what's involved: 0xffA0, 0xffA2, 0xffA1, A, BC, HL, and DE. We know what BC, HL, and DE should contain: that header element which is decreasing every read, the cursor in the compressed data stream, and the cursor in the decompressed data stream, respectively. We read the first byte of the record into 0xffA0 (thru A) then decrement BC and do the same check as earlier. We then read the second byte into 0xffA2 and to the same decrement and check against BC again. We can disasm this to:

```
mem_ffA0 = *compressed++;
if (--head2 == -1) ret;
mem_ffA2 = *compressed++;
if (--head2 == -1) ret;
```



```
ROM0:18DE 2A             ldi  a,(hl)          ;2  24
ROM0:18DF E0 A0          ld   (ff00+A0),a     ;3  27
ROM0:18E1 0B             dec  bc              ;2  29
ROM0:18E2 79             ld   a,c             ;1  30
ROM0:18E3 A0             and  b               ;1  31
ROM0:18E4 3C             inc  a               ;1  32
ROM0:18E5 C8             ret  z               ;2  34
ROM0:18E6 2A             ldi  a,(hl)          ;2  36
ROM0:18E7 E0 A2          ld   (ff00+A2),a     ;3  39
ROM0:18E9 0B             dec  bc              ;2  41
ROM0:18EA 79             ld   a,c             ;1  42
ROM0:18EB A0             and  b               ;1  43
ROM0:18EC 3C             inc  a               ;1  44
ROM0:18ED C8             ret  z               ;2  46
ROM0:18EE C5             push bc              ;4  50
ROM0:18EF F0 A2          ld   a,(ff00+A2)     ;3  53
ROM0:18F1 47             ld   b,a             ;1  54
ROM0:18F2 E6 F0          and  a,F0            ;2  56
ROM0:18F4 CB 37          swap a               ;2  58
ROM0:18F6 E0 A1          ld   (ff00+A1),a     ;3  61
ROM0:18F8 78             ld   a,b             ;1  62
ROM0:18F9 E6 0F          and  a,0F            ;2  64
ROM0:18FB C6 03          add  a,03            ;2  66
ROM0:18FD 47             ld   b,a             ;1  67
ROM0:18FE E5             push hl              ;4  71
ROM0:18FF 21 A0 FF       ld   hl,FFA0         ;3  74
ROM0:1902 7B             ld   a,e             ;1  75
ROM0:1903 96             sub  (hl)            ;2  77
ROM0:1904 4F             ld   c,a             ;1  78
ROM0:1905 23             inc  hl              ;2  80
ROM0:1906 7A             ld   a,d             ;1  81
ROM0:1907 9E             sbc  (hl)            ;2  83
ROM0:1908 67             ld   h,a             ;1  84
ROM0:1909 69             ld   l,c             ;1  85
ROM0:190A 2B             dec  hl              ;2  87
ROM0:190B 2A             ldi  a,(hl)          ;2  89
ROM0:190C 12             ld   (de),a          ;2  91
ROM0:190D 13             inc  de              ;2  93
ROM0:190E 05             dec  b               ;1  94
ROM0:190F 20 FA          jr   nz,190B         ;2  96
ROM0:1911 E1             pop  hl              ;3  99
ROM0:1912 C1             pop  bc              ;3  102
ROM0:1913 C3 AF 18       jp   18AF            ;4  106
```

Next it pushes BC meaning we can assume it's no longer going to be head2 until it's popped again on ROM0:1912, then it restores A from 0xffA2, moves it to B, and then turns A into just the upper nibble. SWAP moves that upper nibble to the lower position. Seems like this second byte is actually regarded as two separate nibbles, then. This nibble is then stored in 0xffA1 and it restores and ensures A as being just the original lower nibble. It adds 3 to this, moves it to B, backs up HL, and sets it to the address of our earlier value stored in 0xffA0 (the first byte of the record). The next part seems a bit obtuse, so let's decompile what we have:

```
a = mem_ffA2;
mem_ffA1 = (a & 0xF0) >> 4;
b = (a & 0x0F) + 3;
```

To me, this next section looks obtuse because it's splitting up DE and acting on it independently. This clearly has a deeper meaning, but we'll have to figure it out. A is currently 0x01 originally

from E and (0xffA0) is currently 0x00 originally from the first record byte. It subtracts these and stores the result in C. Then it increments HL to 0xffA1, loads D into A, and this time does a Subtract with Carry against (0xffA1) which was originally from the upper nibble of the second record byte. So why with carry? Carry is set if the earlier subtraction underflowed. What this means is that this is a 16-bit subtraction. It then stores the result in HL and dereferences it. This means that mem_ffA0 is actually a 12-bit offset instead of 0xffA0 and 0xffA1 being two separate 8-bit values. Nice, this is part of the difference we needed to understand. Before indexing it, it also decrements HL. So let's put this together, adjusting the existing code to match our new info:

```
uint16_t offset = *compressed++;
if (--head2 == -1) ret;
uint8_t mem_ffA2 = *compressed++;
if (--head2 == -1) ret;
a = mem_ffA2;
offset |= (a & 0xF0) << 4;
b = (a & 0x0F) + 3;
hl = decompressed - offset - 1;
```

Finally we're on the final stretch. It copies (HL) into (DE) through A, increments HL and DE, decrements B, then, if B has not reached zero, returns to ROM0:190B, where we copy out of (HL), again. This is a simple loop, so let's implement it:

```
do {
      *decompressed = *hl++;
      decompressed++;
} while (--b);
```

That's it! We've reached the end of the code segment. This is how the repetition record is handled. To summarize, the record is two bytes of the form 0xLLUC where the offset from the head of the decompression stream is 0xULL + 1 and the number of bytes to copy is 0xC + 3.

Next you can implement the rest of the algorithm we know with this decompilation and make sure it works. Then, you're free to make the code nicer. Of course, you can also just grab it from GitHub too.

So what was that table we found? Let's return to ROM0:1360 and take a quick look. It seems like there are many sequences of 0xFFFFFF so it's probably a section end rather than a table end. In order to find where the table starts, compare the bytes with the structure we know until something doesn't match. It should be a series of 6-byte records followed by a 0xFFFFFF terminator, where the second 3-byte entry should look something like 0x009000, that is, be a VRAM reference. If we scroll back, we see highly entropic data around ROM0:10E0 so the next terminator after that is at ROM0:1111. The pattern seems to terminate at ROM0:10F3, as the three bytes before that are not a VRAM address.

Find the tilemap

Now that you've experienced finding the decompression algorithm. Finding the tilemap should be a snap. Can you remember the process? First clear all your breakpoints. Now, here's the high-level overview: in the VRAM viewer go to the BG map tab and look for some tile, mouse-over it and you'll see the address in "Map address". Simply set a on-write access breakpoint on that address and work backwards from where it triggers to find the original address in ROM. Good luck!

How to approach this without the head start

We started this tutorial with a decent idea of how the algorithm worked thanks to abridgewater. But what if we didn't? How would this be different? We'd start the same way and find the algorithm in the same way, of course. However, we likely would've had to have reversed the entire algorithm. In my opinion, that's the easiest method once we're sure we've found the algorithm.

Once you rewrite the algorithm in some script language, extract a test case. That is, extract some compressed data (from ROM) and its corresponding decompressed data (from RAM), then run your tool over the compressed data and see if the output matches what you pulled from RAM. If it does, then you can consider writing a reverse of the algorithm in order to make a compressor, but this is rather difficult. The easiest way to make a compressor is to figure out what algorithm it is, then find an existing implementation to copy and adjust for your needs. In this case, the algorithm is LZ77.

Credits and thanks

- Author: YasaSheep
- abridgewater for the initial differential analysis of the algo, which made this work a lot easier.
- Bunkai for starting this tutorial series.
- The RHDN community and all those involved in discussions of this ROM, such as: 256, 343, Bootleg Porygon, Lusofonia, Pluvius, and TF1945.