This tutorial is meant as a personal reminder for how I accomplished a few cheat tasks in Ubuntu 17.04 using the 32-bit versions of HxD, Cheat Engine 6.51, and latest nightly of PPSSPP.

These are the code types I'm interested in: (reference http://raing3.gshi.org/psp-utilities/?action=page&file=PSP/CodeTypes https://datacrystal.romhacking.net/wiki/CwCheat#Multi-Write_.280x4.2C_0x8.29)

Chapter 1 Data Structures

Constant RAM Writes	
Type 0x00 8-bit 0XXXXXXX YYYYYYYY	Writes byte YY to [XXXXXXX].
Type 0x01 16-bit 1XXXXXXX 0000YYYY	Writes halfword YYYY to [XXXXXXX].
Type 0x02 32-bit 2XXXXXXX 000000YY	Writes word YYYYYYYY to [XXXXXXX].

Type 0x0D Joker Code D00000YY 1XXXXXXX	Checks if (ctrl & XXXXXXX) == XXXXXXX. If not, the next YY+1 lines are not executed (ie. execution status is set to false for YY+1 lines).

PSP_CTRL_SELECT	0x00000001
PSP_CTRL_START	0x00000008
PSP_CTRL_UP	0x00000010
PSP_CTRL_RIGHT	0x00000020
PSP_CTRL_DOWN	0x00000040
PSP_CTRL_LEFT	0x00000080
PSP_CTRL_LTRIGGER	0x00000100
PSP_CTRL_RTRIGGER	0x00000200
PSP_CTRL_TRIANGLE	0x00001000
PSP_CTRL_CIRCLE	0x00002000
PSP_CTRL_CROSS	0x00004000
PSP_CTRL_SQUARE	0x00008000

Type 0x04 32-bit Multi Write 4XXXXXXX YYYYZZZZ VVVVVVVV WWWWWWWW	Starting at address [XXXXXXX], this code will loop YYYY times. The next address is determined by the incrementing the current address by (ZZZZ * 4). The value written to the address is specified by VVVVVVVV+ (WWWWWWWW * loop count).

_L 0x4AAAAAAA 0xNNNNSSSS _L 0xVVVVVVVV 0xXXXXXXXX	Multi-write word AAAAAAA = where to start writing VVVVVVVV = first value to write NNNN = number of values to write SSSS * 4 = offset to add to the address after writing each value XXXXXXXX = value to add to the value after writing each value
_L 0x8AAAAAAA 0xNNNNSSSS _L 0x000000VV 0x000000XX	Multi-write byte AAAAAAA = where to start writing VV = first value to write NNNN = number of values to write SSSS = offset to add to the address after writing each value XX = value to add to the value after writing each value
_L 0x8AAAAAAA 0xNNNNSSSS _L 0x1000VVVV 0x0000XXXX	Multi-write halfword AAAAAAA = where to start writing VVVV = first value to write NNNN = number of values to write SSSS * 2 = offset to add to the address after writing each value XXXX = value to add to the value after writing each value

Definitions are useless without examples:

_C0 200 Orbs
_L 0x103EA4E0 0x000000C8
_L 0x103EA4E4 0x000000C8
_L 0x103EA4E8 0x000000C8
_L 0x103EA4EC 0x000000C8
_L 0x103EA4F0 0x000000C8
_L 0x103EA4F4 0x000000C8
_L 0x103EA4F8 0x000000C8

_C0 Max Elemental Orbs
_L 0x203EA4E0 0x000003E7
_L 0x203EA4E4 0x000003E7
_L 0x203EA4E8 0x000003E7
_L 0x203EA4EC 0x000003E7
_L 0x203EA4F0 0x000003E7
_L 0x203EA4F4 0x000003E7
_L 0x203EA4F8 0x000003E7

The first thing to notice here is the address for the cheat 3EA4E0. It's the same in both examples. The 200 orbs version shows the address preceded by a "1" 103EA4E0 while the "Max" version is preceded by a "2". This could have also been written with a "0", like this: _L 0x003EA4E0 0x000000C8.

The second thing to note is the value to the right of the address which differs as either 0x000000C8, or 0x000003E7.

This code describes the address to be written on the left, 03EA4E0, and the value to be written on the right 000000C8. The address is absolute but we can fiddle with the value.

To understand the value, consider that 8 bits make up a byte which can be expressed in binary as 00000000 to 11111111, decimal as 0 to 255, or hexadecimal as 00 to FF. 0x000000C8 describes 4 hexadecimal bytes "00 00 00 C8"

So we have an address and value but what is the first number that I keep truncating? To illustrate I'll add one more level of complication. This code: _L 0x103EA4E0 0x000000C8 can be expressed thusly: _L 0x003EA4E0 0x000000C8.

I've high-lighted various sections in red, green and blue. The red portion, or first number of the address, is not part of the address. It describes how the value will be written:

0 writes C8 – one byte, 8 bits
1 writes 00C8 – two bytes (half-word), 16 bits
2 writes 000000C8 – four bytes (word), 32 bits

Data alignment is of critical importance for the MIPS architecture:

0 – byte, no alignment required
1 – half-word, must be aligned to even hex address: 0, 2, 4, 6, 8, A, C, E
2 – word, must be aligned to hex multiple of 4: 0, 4, 8, C

With code alignment in mind, it's crucial to only write as much data, the value, as needed. To do that we'll need to count out the address so I'll change my example slightly.

_L 0x203EA4E0 0x58575655:

3EA4E0 - 55
3EA4E1 - 56
3EA4E2 - 57
3EA4E3 - 58

It’s a type “0x2” code which will write all 8 hex digits. Each set of two digits goes to a particular address. Address 3EA4E0 gets the far right value of 55 and, with each subsequent pair counted inward, we’ll add “1” to the address. Looking at the result we can see the next address will be 3EA4E4 and what do we find in the actual code?

_L 0x203EA4E0 0x000003E7
_L 0x203EA4E4 0x000003E7

Now I said we could express the one code this way, _L 0x003EA4E0 0x000000C8 and it’ll look like this:

3EA4E0 - C8
3EA4E1 - no write (skip)
3EA4E2 - no write (skip)
3EA4E3 - no write (skip)

Or this, _L 0x103EA4E0 0x000000C8

3EA4E0 - C8
3EA4E1 - 00
3EA4E2 - no write (skip)
3EA4E3 - no write (skip)

Or this, _L 0x203EA4E0 0x000000C8

3EA4E0 - C8
3EA4E1 - 00
3EA4E2 - 00
3EA4E3 - 00

If it doesn’t look like a big difference, think again. If we only need to write one single byte (hex pair) then we should be using a type “0x0” code. Using a “0x2” will force 00’s into addresses that maybe shouldn’t be touched.

Based on the example we should definitely re-write that one code this way:

_C0 200 Orbs
_L 0x003EA4E0 0x000000C8
_L 0x003EA4E4 0x000000C8
_L 0x003EA4E8 0x000000C8
_L 0x003EA4EC 0x000000C8
_L 0x003EA4F0 0x000000C8
_L 0x003EA4F4 0x000000C8
_L 0x003EA4F8 0x000000C8

And while we're at it, let's re-write the whole thing using a multi-write cheat like so:

_C0 Max Elemental Orbs 8-bit
_L 0x803EA4E0 0x00070004
_L 0x000000C8 0x00000000

_C0 Max Elemental Orbs 16-bit
_L 0x803EA4E0 0x00070002
_L 0x100000C8 0x00000000

_C0 Max Elemental Orbs 32-bit
_L 0x403EA4E0 0x00070001
_L 0x000000C8 0x00000000

Oh boy! Firstly I've shown two type "0x8" multi-writes and one type "0x4". The former keys off of the first digit on the second line in a way we're used to seeing, as either type "0x0" 8-bit, or "0x1" 16-bit. Type "0x4" is a 32-bit code so it doesn't need to describe a type "0x2".

Once again, each example shows a different way to write C8 - as either itself, 00C8 or 000000C8. The other line I'm going to focus on is written three ways 0x00070004, 0x00070002, and 0x00070001. It says we're going to write 1, 2 or 4 bytes (by steps of 4, 2, or 1) 7 times.

The 8-bit version (0x00070004), out of 4 (1 byte) steps, writes 1 byte (skipping 3), 7 times. To get the end address we just need to do the hex math: 4 x 1 x 7 + 3EA4E0 = 3EA4FC

3EA4FC is the next address to write. The last address written is -1 (minus skip).

The 16-bit version (0x00070002), out of 2 (2 byte) steps writes 2 bytes (skipping 2), 7 times. To get the end address we just need to do the hex math: 2 x 2 x 7 + 3EA4E0 = 3EA4FC

The 32-bit version (0x00070001), out of 1 (4 byte) step writes 4 bytes (skipping none), 7 times. To get the end address we just need to do the hex math: 1 x 4 x 7 + 3EA4E0 = 3EA4FC

Phew! Okay, we re-wrote a seven line code three different ways using only two lines of code. Yay! Which method is the best one to use? That's going to be determined by the size of the value, (byte, half-word, word) the repetition, and word alignment. To get a better idea of this lets look at a different example:

_C0 Item All A
_L 0x408077B0 0x00070001
_L 0x0A0A0A0A 0x00000000
_L 0x208077CC 0x010A0A0A

_C0 All Items
_L 0x808077B0 0x001F0001
_L 0x0000000A 0x00000000
_L 0x008077CF 0x00000001

Believe it or not, both codes describe the same thing.

The “0x4” is: 4 x 1 x 7 + 8077B0 = 8077CC
_L 0x208077CC 0x010A0A0A

The “0x8” is: 1 x 1 x 1F + 8077B0 = 8077CF
_L 0x008077CF 0x00000001

Um… That’s not the same. Really? When you look at the next line of the type “0x4” code it starts at address 8077CC and writes values:

8077CC - 0A
8077CD - 0A
8077CE - 0A
8077CF - 01

While the type “0x8” code writes 01 to address 8077CF. Both codes accomplish the same thing in a different way so which is better?

My preference is the code that has the least duplication. The 8-bit code is flexible enough to write out just as many 0A values we’d like and stop exactly where we want while the 32-bit code requires finagling. Speaking of which, these two are the same as well:

_C0 Item All A
_L 0x408077B0 0x00070001
_L 0x0A0A0A0A 0x00000000
_L 0x208077CC 0x010A0A0A

_C0 Item All A
_L 0x808077B0 0x000F0001
_L 0x10000A0A 0x00000000
_L 0x108077CE 0x0000010A

The “0x8” is: 2 x 1 x F + 8077B0 = 8077CE
_L 0x108077CE 0x0000010A

While confusing, the important thing to notice is the foundational structure and how it’s applied. After all, it is just a matter of time until you see something like this:

_L 0x002EECBA 0x000000E7
_L 0x002EECBB 0x00000003
_L 0x002EECBC 0x000000E7
_L 0x002EECBD 0x00000003
_L 0x002EECBE 0x000000E7
_L 0x002EECBF 0x00000003
_L 0x002EECC0 0x000000E7
_L 0x002EECC1 0x00000003

Which can be expressed as:
_L 0x102EECBA 0x000003E7
_L 0x102EECBC 0x000003E7
_L 0x102EECBE 0x000003E7
_L 0x102EECC0 0x000003E7

Which can be expressed as:
_L 0x202EECBA 0x03E703E7
_L 0x202EECBE 0x03E703E7

Which can be expressed as:
_L 0x802EECBA 0x00040001
_L 0x100003E7 0x00000000

And while all of the afore expressions are literally correct, did you notice which one is wrong?

It's this one:
_L 0x202EECBA 0x03E703E7
_L 0x202EECBE 0x03E703E7

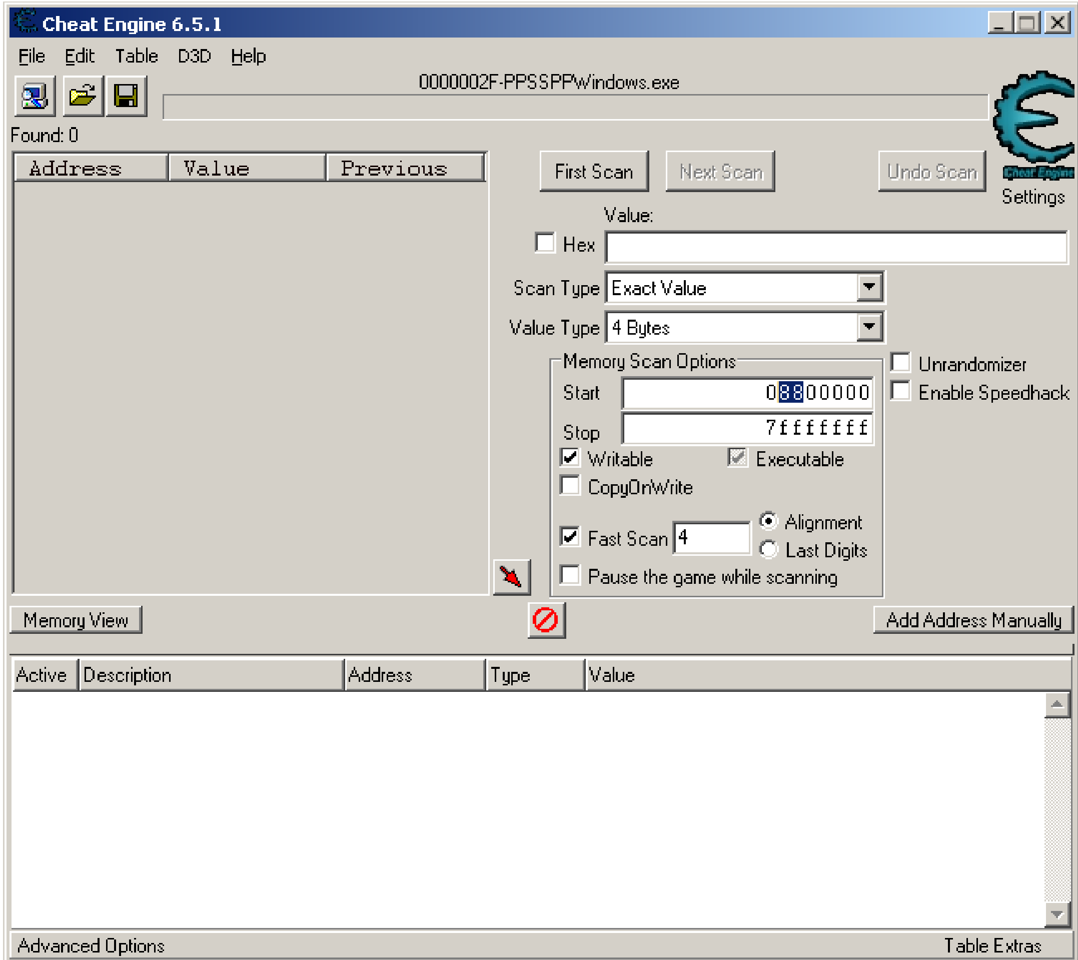
Type "0x2" 32-bit codes have to be aligned to an address that is a multiple of 4 which is 0, 4, 8, C. PPSSPP's built in implementation of the CWCheat engine may correct this but don't expect CWCheat or TempAR, running on actual hardware, to. A corrected version of the mis-aligned code is shifted like so:

_L 0x102EECBA 0x000003E7
_L 0x202EECBC 0x03E703E7
_L 0x102EECC0 0x000003E7

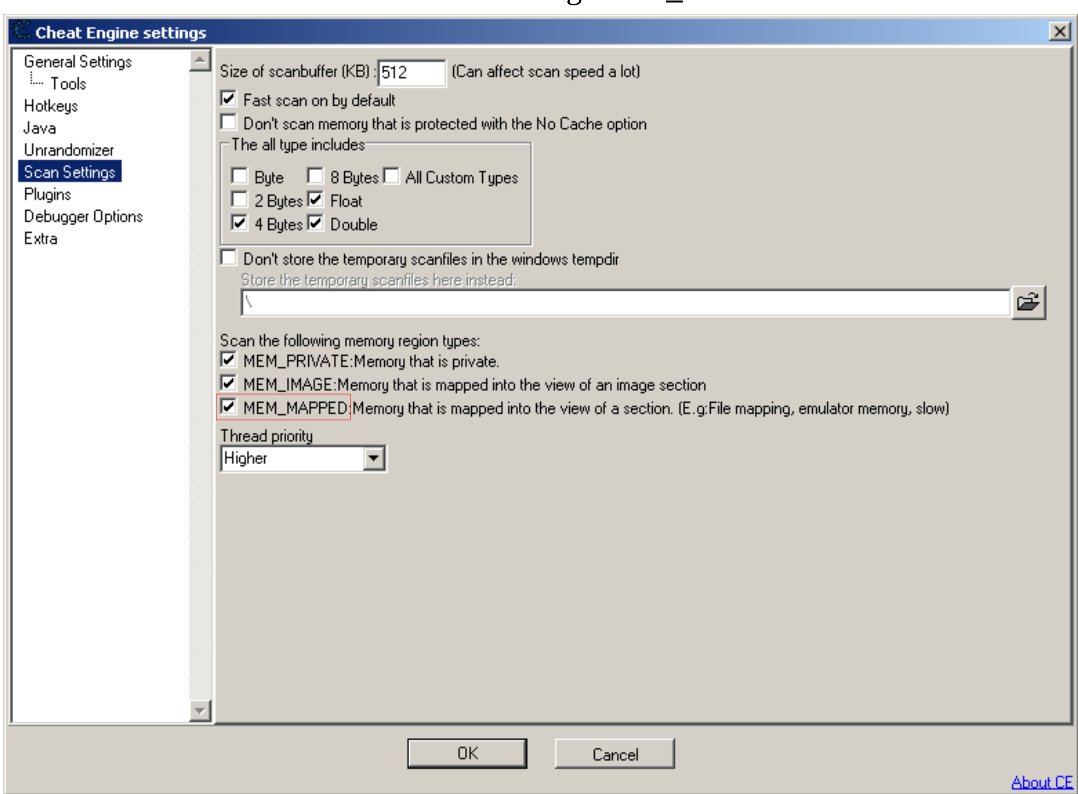
The multi-write code avoids the problem because it's a 16-bit code that only needs to be aligned to an even address.

Chapter 2 Tools

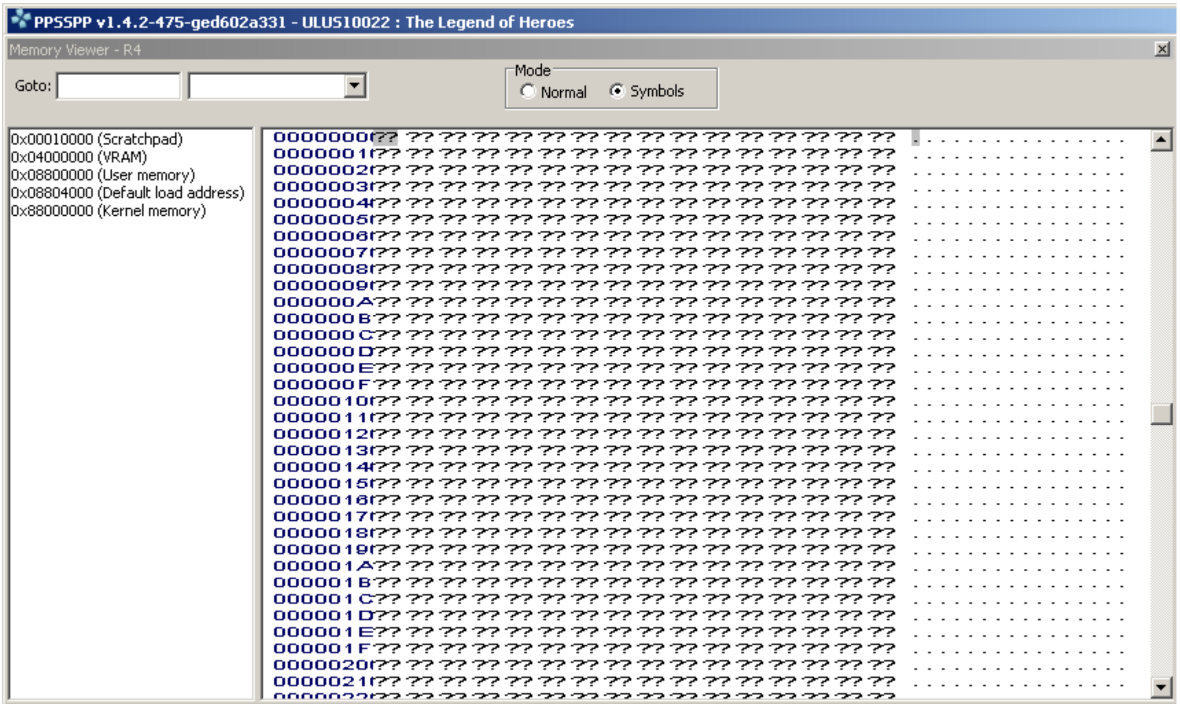
This chapter is going to be about using PPSSPP and Cheat Engine. It's recommended that you run PPSSPP, then attach Cheat Engine to the process. Best to set the Memory Scan Options Start to 08800000.



And make sure the emulation setting MEM_MAPPED is enabled.



Remember those addresses I was talking about before, it's time to find them. PPSSPP is pretty clear cut about it when you look at the memory view (CTRL+M).

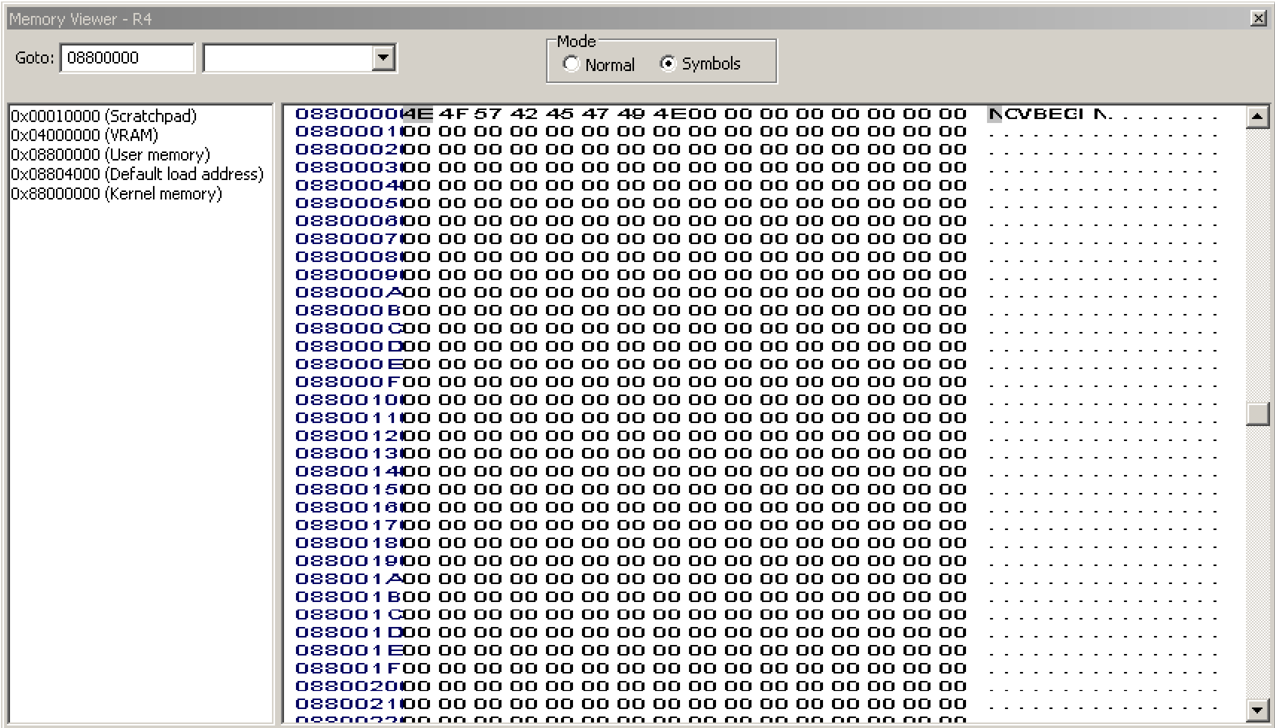


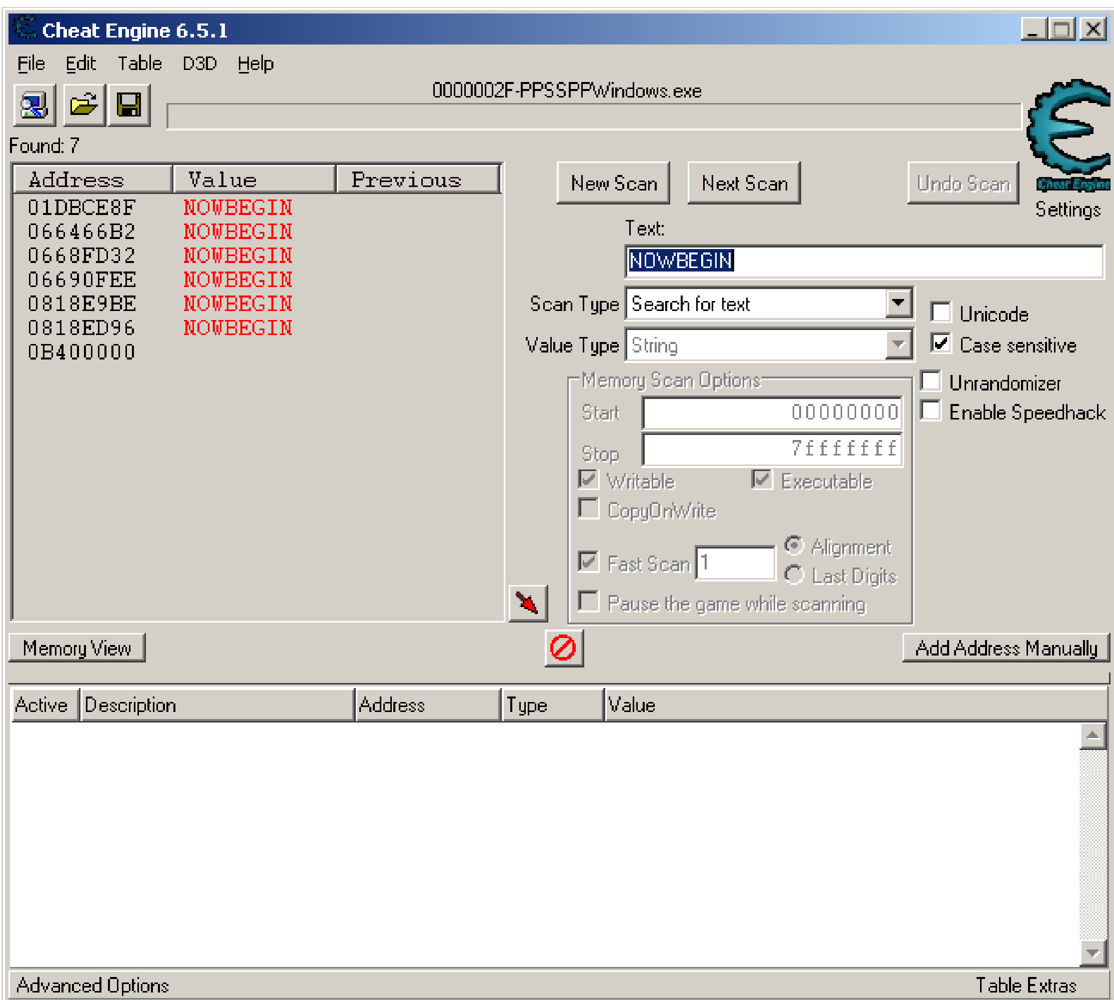
User memory starts at 0x08800000 and that's where the cheat codes lie. I actually have a cheat that'll help to illustrate and smooth things with Cheat Engine:

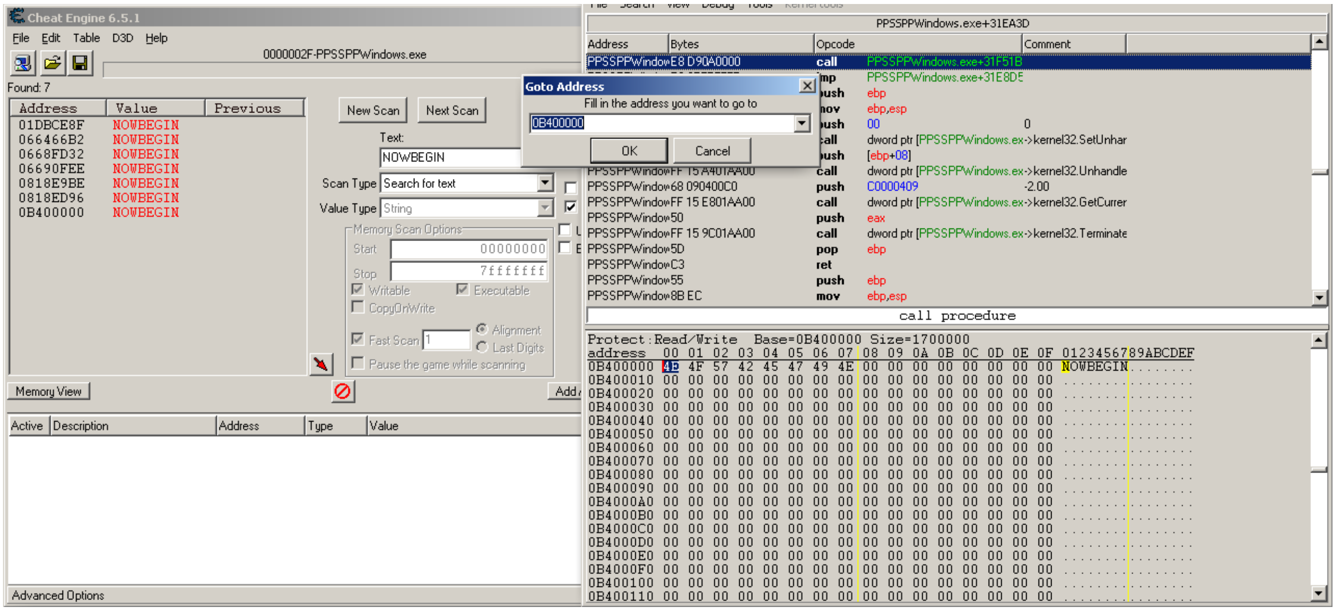
_C0 CHTENG String NOWBEGIN
_L 0x20000000 0x00000000
_L 0x20000004 0x00000000
_L 0xD0000001 0x10008000
_L 0x20000000 0x42574F4E
_L 0x20000004 0x4E494745

Notice that the starting address, in CWCheat format, is 0000000 but we already said we're going to be going to 8800000. **CWCheat formatted codes must have 8800000 added to them in order to find the correct address.**

Like any cheat code, this one only works when the game is running in PPSSPP and then, only while the "Square" button is pressed on the controller. This is what it does:


Memory Viewer - R4
Goto: 08800000
Mode
Normal
Symbols
0x00010000 (Scratchpad)
0x04000000 (VRAM)
0x08800000 (User memory)
0x08804000 (Default load address)
0x88000000 (Kernel memory)
08800000 4E 4F 57 42 45 47 49 4E 00 00 00 00 00 00 00 00 NCVBEGI N.
08800010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08800020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08800030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08800040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08800050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08800060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08800070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08800080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08800090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
088000A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
088000B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
088000C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
088000D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
088000E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
088000F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08800100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08800110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08800120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08800130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08800140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08800150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08800160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08800170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08800180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08800190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
088001A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
088001B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
088001C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
088001D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
088001E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
088001F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08800200 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08800210 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00


Cheat Engine 6.5.1
File
Edit
Table
D3D
Help
0000002F-PPSSPPWindows.exe
Found: 7
Address
Value
Previous
01DBCE8F NOWBEGIN
066466B2 NOWBEGIN
0668FD32 NOWBEGIN
06690FEE NOWBEGIN
0818E9BE NOWBEGIN
0818ED96 NOWBEGIN
0B400000
New Scan
Next Scan
Undo Scan
Settings
Text:
NOWBEGIN
Scan Type
Search for text
Value Type
String
Unicode
Case sensitive
Memory Scan Options
Start
00000000
Stop
7fffffff
Writable
Executable
CopyOnWrite
Fast Scan
1
Alignment
Last Digits
Pause the game while scanning
Unrandomizer
Enable Speedhack
Memory View
Add Address Manually
Active
Description
Address
Type
Value
Advanced Options
Table Extras



The first screen is the PPSSPP Memory Viewer set to 08800000 with Square pressed. You can see the words “NOWBEGIN”. The following screen shows where I set Cheat Engine (CE) to do a String search for those words but I released the Square button after searching. The last screen shows CE with its memory viewer open and the Square button pressed (CTRL+M, Right click in lower part of Memory Viewer window, select Goto Address, use 0B400000).

My code merely facilitates finding the start memory range in CE. PPSSPP, as an emulator, knows where it’s mapping the virtual PSP memory system. CE doesn’t know anything about that, it sees all of the memory that PPSSPP uses so we need a little trick to find the starting point.

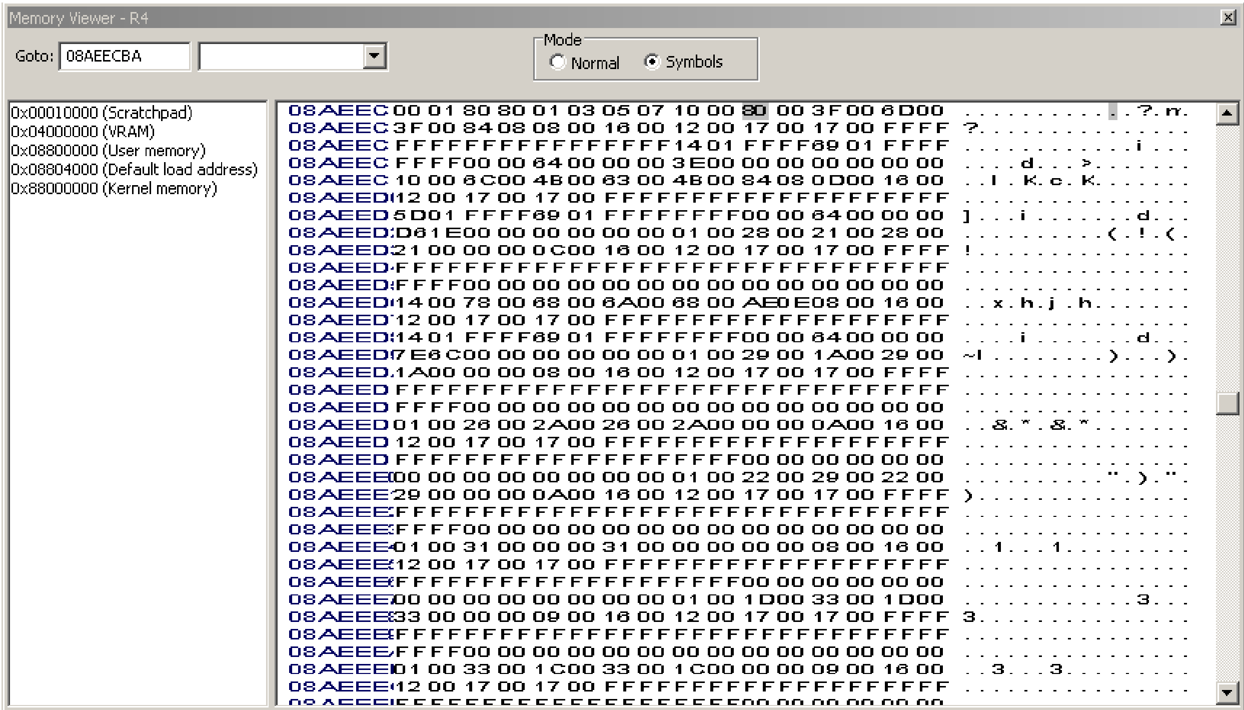
Notice I set the CE Start to 00000000 and it found a bunch of “mirrors” for “NOWBEGIN” but the moment I released the Square button, my cheat code reset back to 0’s (which was reflected in the second screen). The real beginning address is easy to find using this trick.

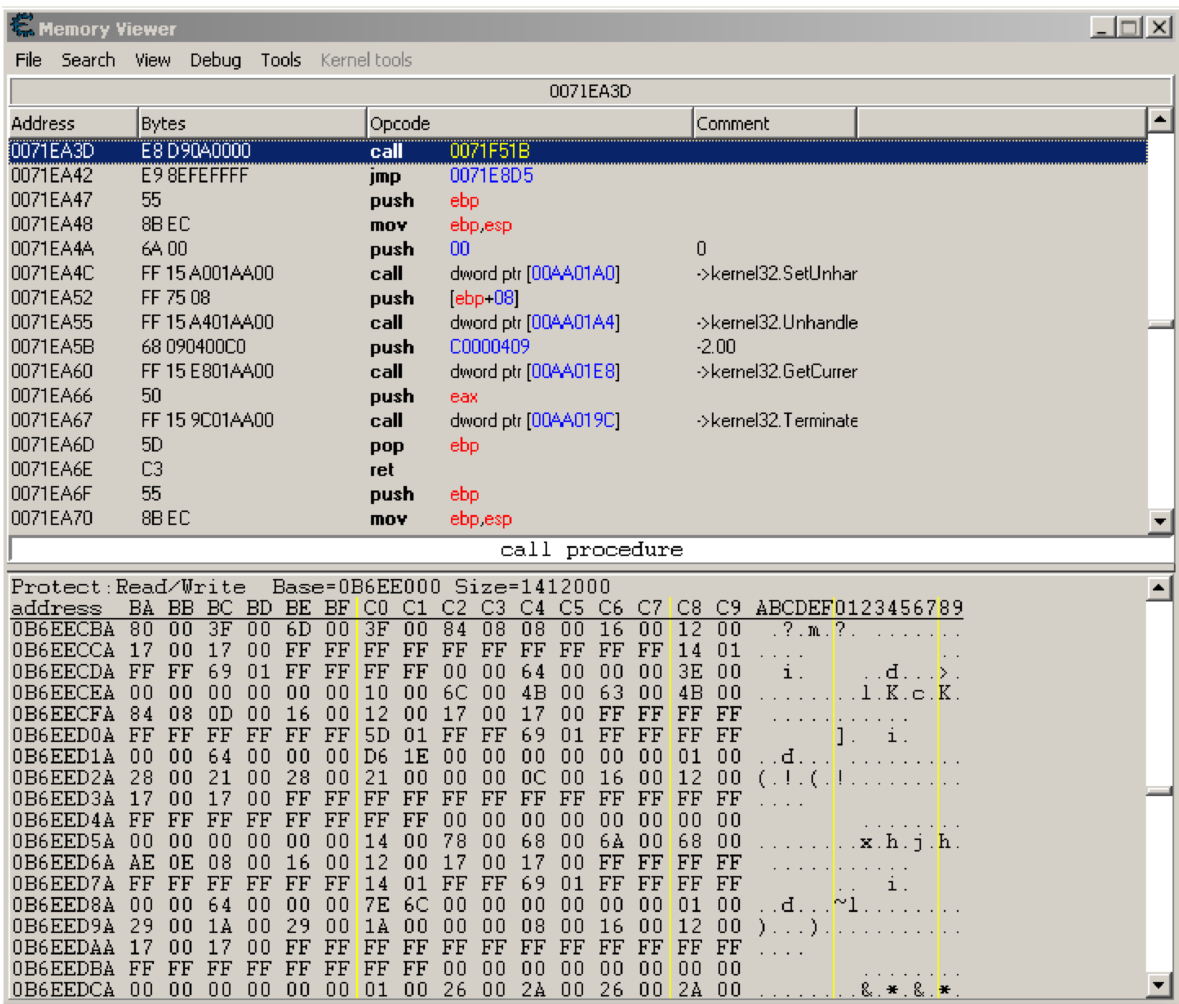
To summarize, CWCheat address 0000000 = PPSSPP (PSP) address 08800000 = (in this case) CE 0B400000.

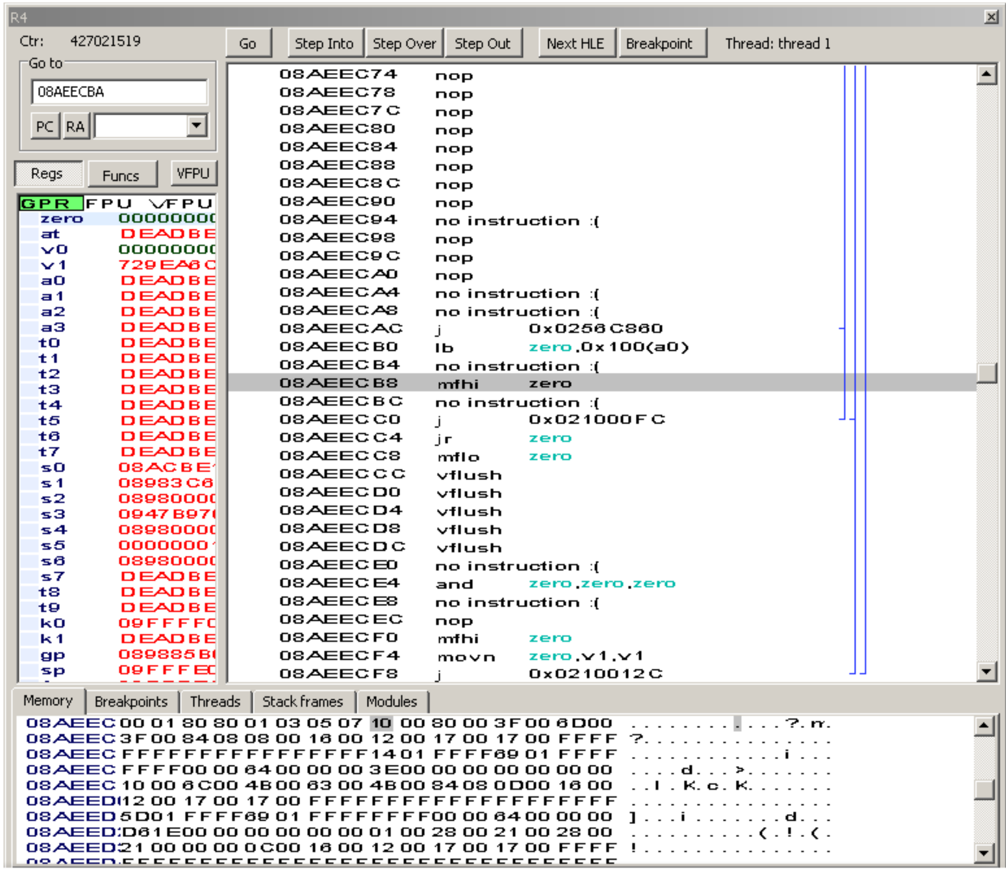
From our last example of chapter one:
_L 0x802EECBA 0x00040001
_L 0x100003E7 0x00000000

The CWCheat address is 2EECBA, PPSSPP is 08800000 + 2EECBA = 08AEECBA, CE is 0B400000 + 2EECBA = 0B6EECBA.

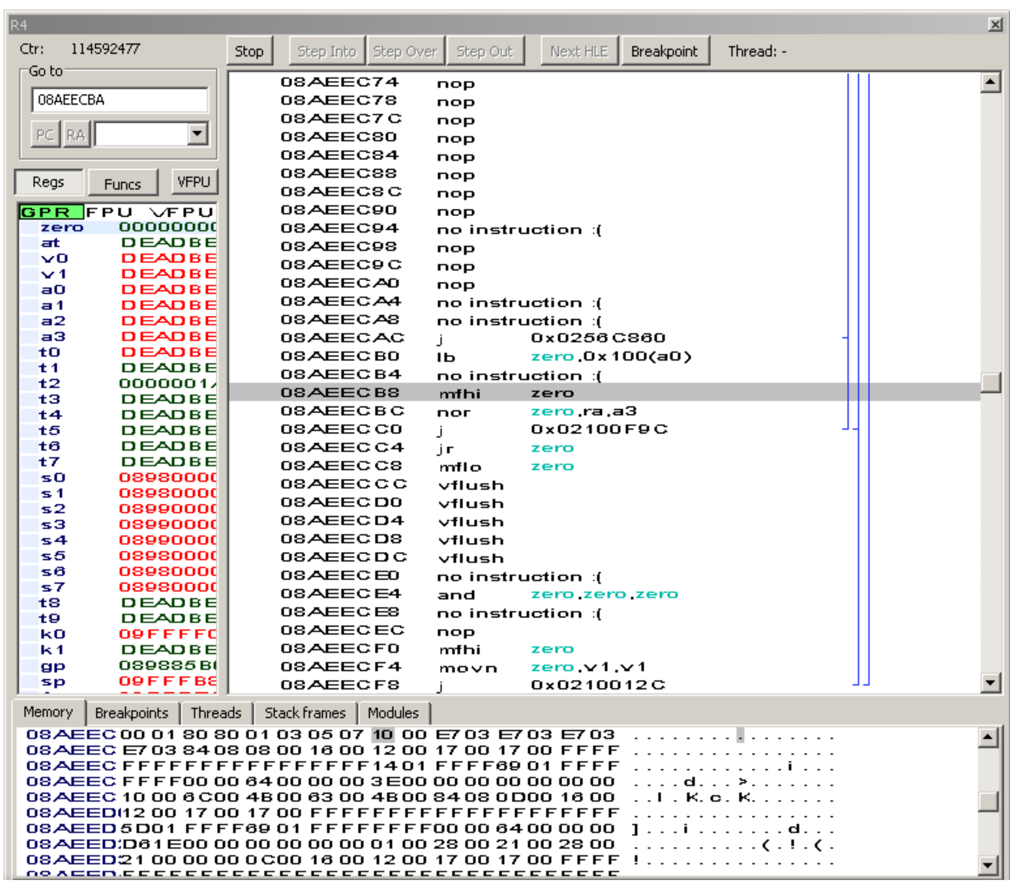
The following two screens show this and it is easy to discern that the patterns are the same. The third screen shows PPSSPP’s Disassembler (CTRL+D) set to the appropriate address.


Memory Viewer - R4
Goto: 08AEECBA
Mode
Normal
Symbols
0x00010000 (Scratchpad)
0x04000000 (VRAM)
0x08800000 (User memory)
0x08804000 (Default load address)
0x88000000 (Kernel memory)


Memory Viewer
File Search View Debug Tools Kernel tools
0071EA3D
Address Bytes Opcode Comment
0071EA3D E8 D90A0000 call 0071F51B
0071EA42 E9 8EFEFFFF jmp 0071E8D5
0071EA47 55 push ebp
0071EA48 8B EC mov ebp,esp
0071EA4A 6A 00 push 00 0
0071EA4C FF 15 A001AA00 call dword ptr [00AA01A0] ->kernel32.SetUnhar
0071EA52 FF 75 08 push [ebp+08]
0071EA55 FF 15 A401AA00 call dword ptr [00AA01A4] ->kernel32.Unhandle
0071EA5B 68 090400C0 push C0000409 -2.00
0071EA60 FF 15 E801AA00 call dword ptr [00AA01E8] ->kernel32.GetCurrer
0071EA66 50 push eax
0071EA67 FF 15 9C01AA00 call dword ptr [00AA019C] ->kernel32.Terminate
0071EA6D 5D pop ebp
0071EA6E C3 ret
0071EA6F 55 push ebp
0071EA70 8B EC mov ebp,esp
call procedure
Protect:Read/Write Base=0B6EE000 Size=1412000



I’m going to enable the cheat so we can see the change.



Comparing the before and after disassembly reveals a subtle change that's only visible by looking at the numbers in the memory portion of the window. Two positions to the right of the grayed cursor block shows pattern E7 03 E7 03 E7 03 E7 03.

The disassembler actually keyed into address 08AEECB8 and that is what's high-lighted. Why is that? Instructions are shown 32-bit so they align at 0, 4, 8, C, without exception. Our code starts at 08AEECBA so it's two positions to the right.

Okay, but why are the values backward? They aren't. The addresses count out just as we saw in chapter 1. The values are little-endian and that is how they are displayed.

Fine, then why is the instruction shown by the disassembler for address 08AEECB8 "mfhi zero"? I don't know MIPS assembly so the instructions are mostly gibberish to me. My assembly knowledge extends to simple operations like jump and nop.

A thing to keep in mind is that we are changing a value in memory that simply keeps track of the character's HP/MP. It's a value, not an instruction. But, since we're talking about PPSSPP's disassembler, it is going to treat any, and every, value as an instruction whether it really is or not.

Chapter 3 Code Porting

In the previous chapter we familiarized with some of the tools used in the cheat trade. In this segment we'll also introduce HxD. Let's dive right in with a code example.

_S ULJS-00016
_C0 Move x2 Hold Circle Norm
_L 0x20069AA8 0x0A200400
_L 0x20001000 0x00E03021
_L 0x20001004 0x0A21A6AC
_L 0x20001008 0x00129042
_L 0xD02CBE65 0x20000020
_L 0x20001008 0x00000000

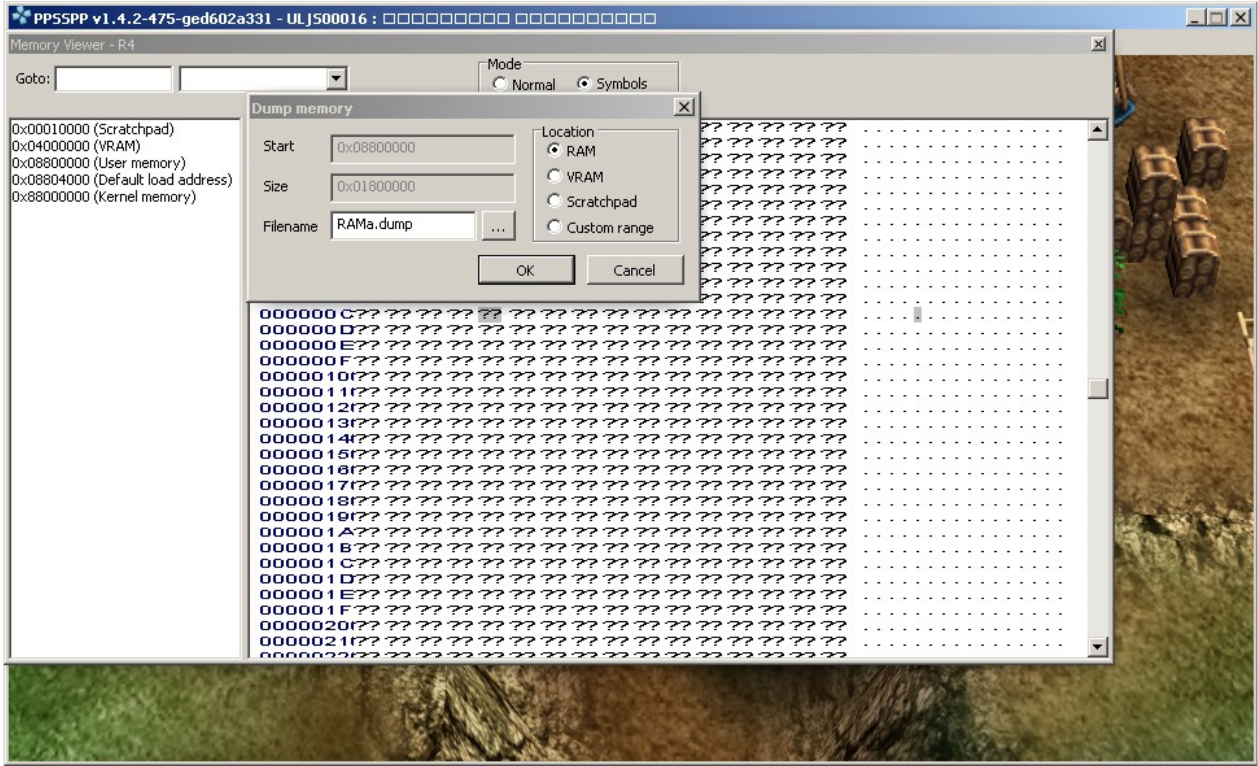
The address high-lighted in red, 0001000, is what you should immediately notice as odd. The PPSSPP Memory Viewer, in chapter 2, said that User Memory starts at 08800000 and the Default Load Address (where PSP code begins) starts at 08804000. CWCheat 0001000 = 08801000 so this address lies within User Memory.

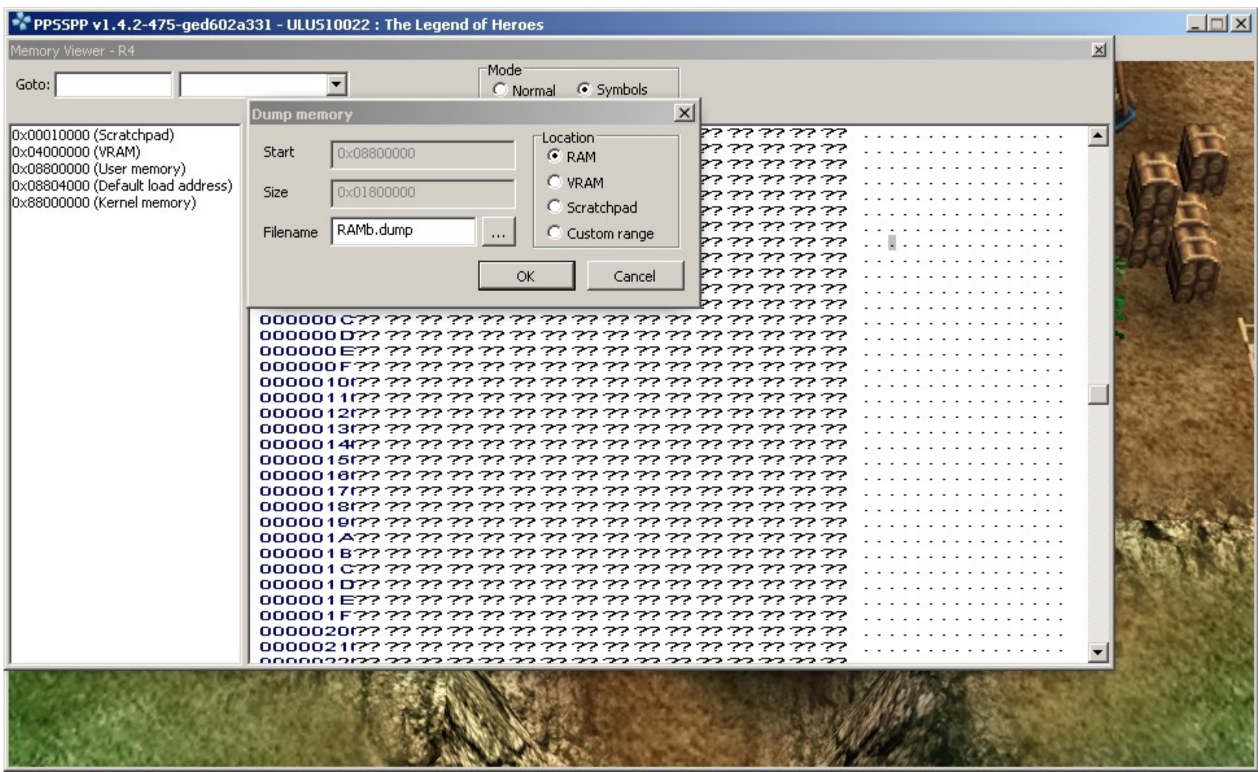
From this, we can infer that value 0A200400 is a jump instruction which takes us into User Memory at 0001000, where more instructions will be placed to alter program execution. Since we're jumping out of the main program flow, it's a pretty good guess that 0A21A6AC is a jump return to get back. The last value of 00129042 is quirky. It occurs after the return jump and is something like read-ahead code execution. Essentially MIPS pops the next instruction onto the stack and that is the last one to be executed.

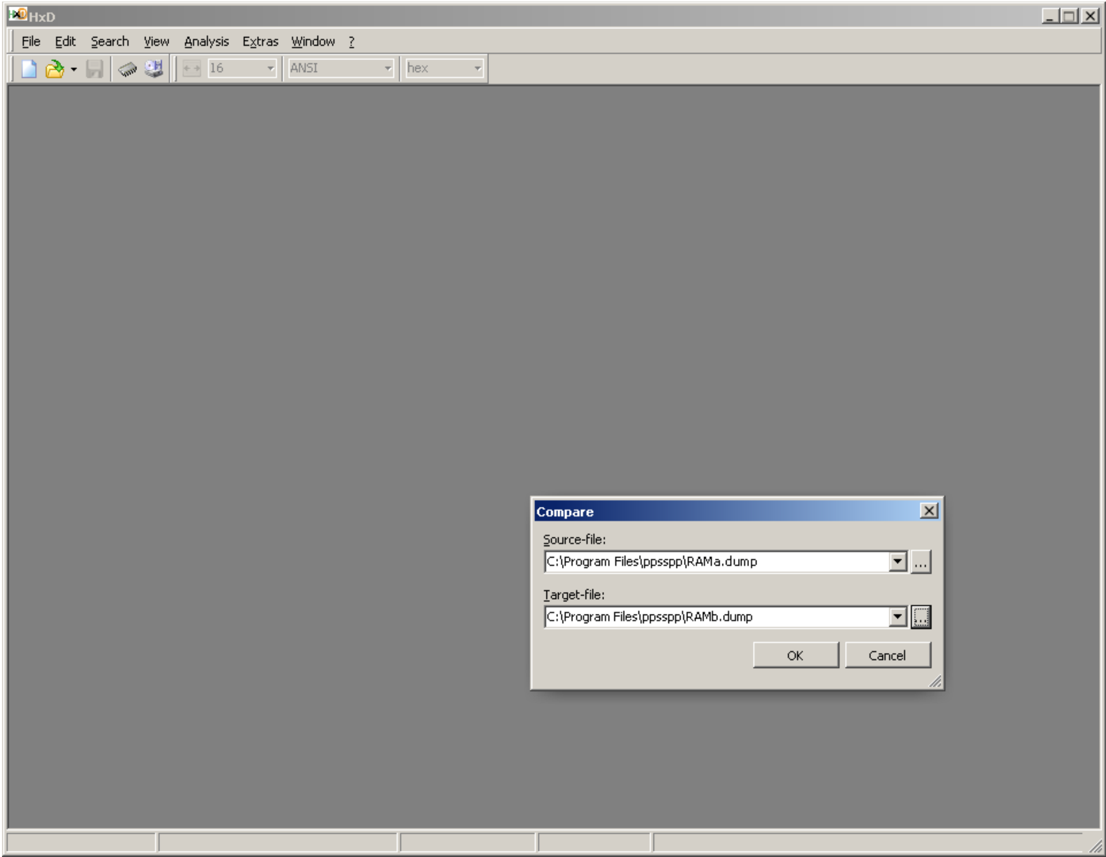
The D02CBE65 address hooks controller events (the Circle button) and, conditionally, executes the following line of _L 0x20001008 0x00000000. For the purpose of this tutorial, we can ignore those last two lines.

Since this chapter is about porting codes, we have to know the game with a code we want, and the game that it's being converted for. In this case it's from _S ULJS-00016 to _S ULUS-10022. We'll need to load both games into PPSSPP, start a new game, then do a Memory Dump at the very first moment we gain control of the character.

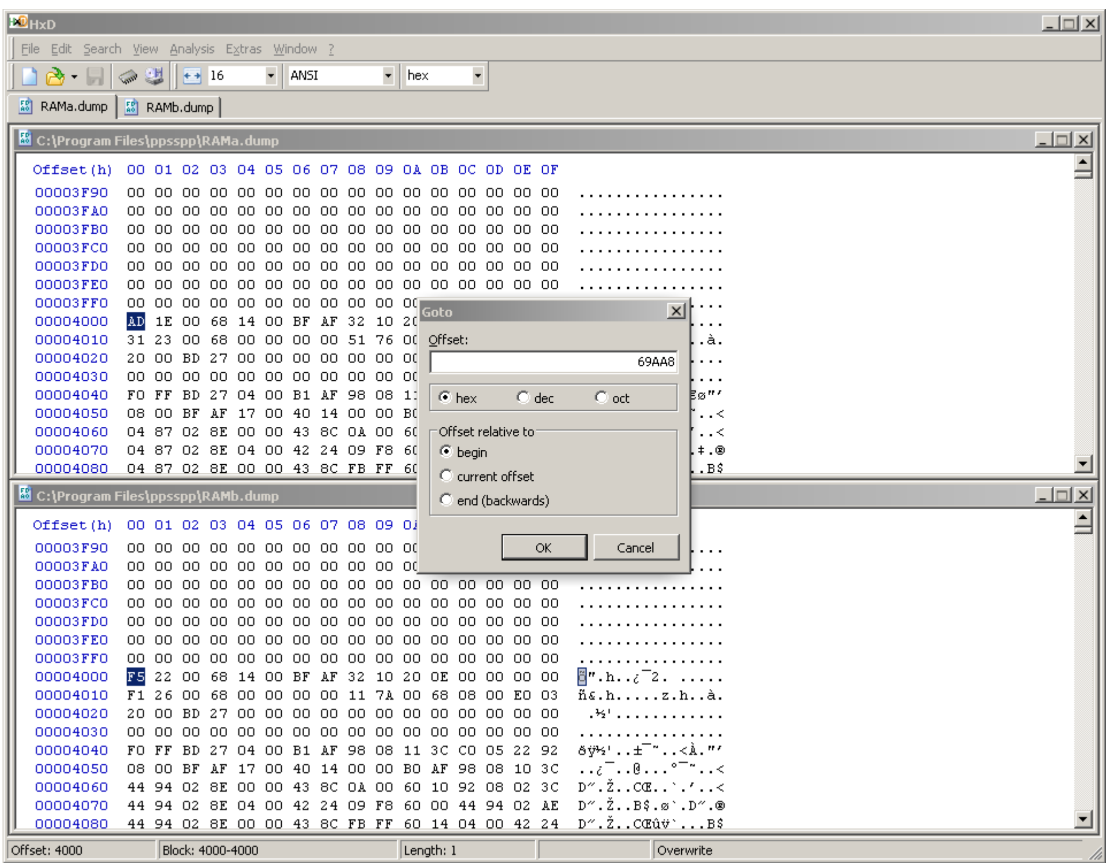
Best practice is to just hit F8 to pause PPSSPP, CTRL+M to bring up the Memory Viewer, right-click anywhere within the window and select Dump. I like to save the source game as RAMa.dump and the target game as RAMb.dump.


PPSSPP v1.4.2-475-ged602a331 - ULJS00016 : □□□□□□□□□ □□□□□□□□□□
Memory Viewer - R4
Goto:
Mode
Normal
Symbols
0x00010000 (Scratchpad)
0x04000000 (VRAM)
0x08800000 (User memory)
0x08804000 (Default load address)
0x88000000 (Kernel memory)
Dump memory
Start
0x08800000
Size
0x01800000
Filename
RAMa.dump
...
Location
RAM
VRAM
Scratchpad
Custom range
OK
Cancel

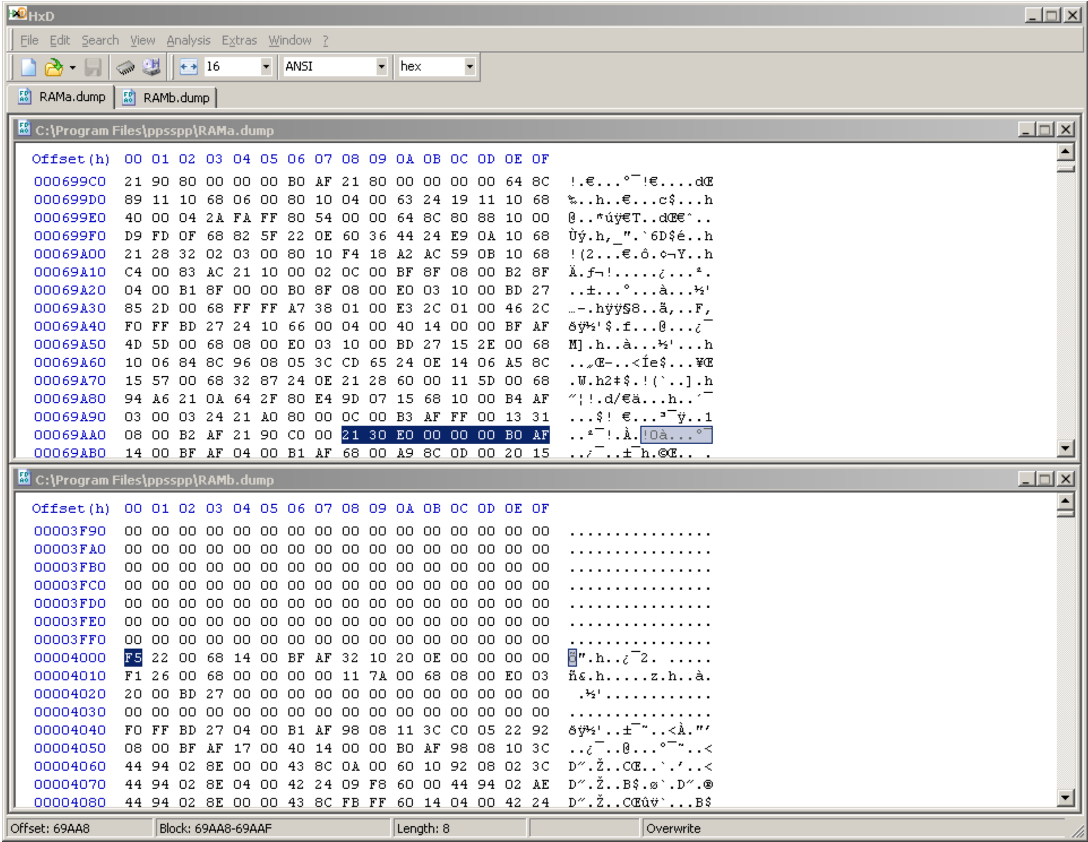

PPSSPP v1.4.2-475-ged602a331 - ULUS10022 : The Legend of Heroes
Memory Viewer - R4
Goto:
Mode
Normal
Symbols
0x00010000 (Scratchpad)
0x04000000 (VRAM)
0x08800000 (User memory)
0x08804000 (Default load address)
0x88000000 (Kernel memory)
Dump memory
Start
0x08800000
Size
0x01800000
Filename
RAMb.dump
...
Location
RAM
VRAM
Scratchpad
Custom range
OK
Cancel



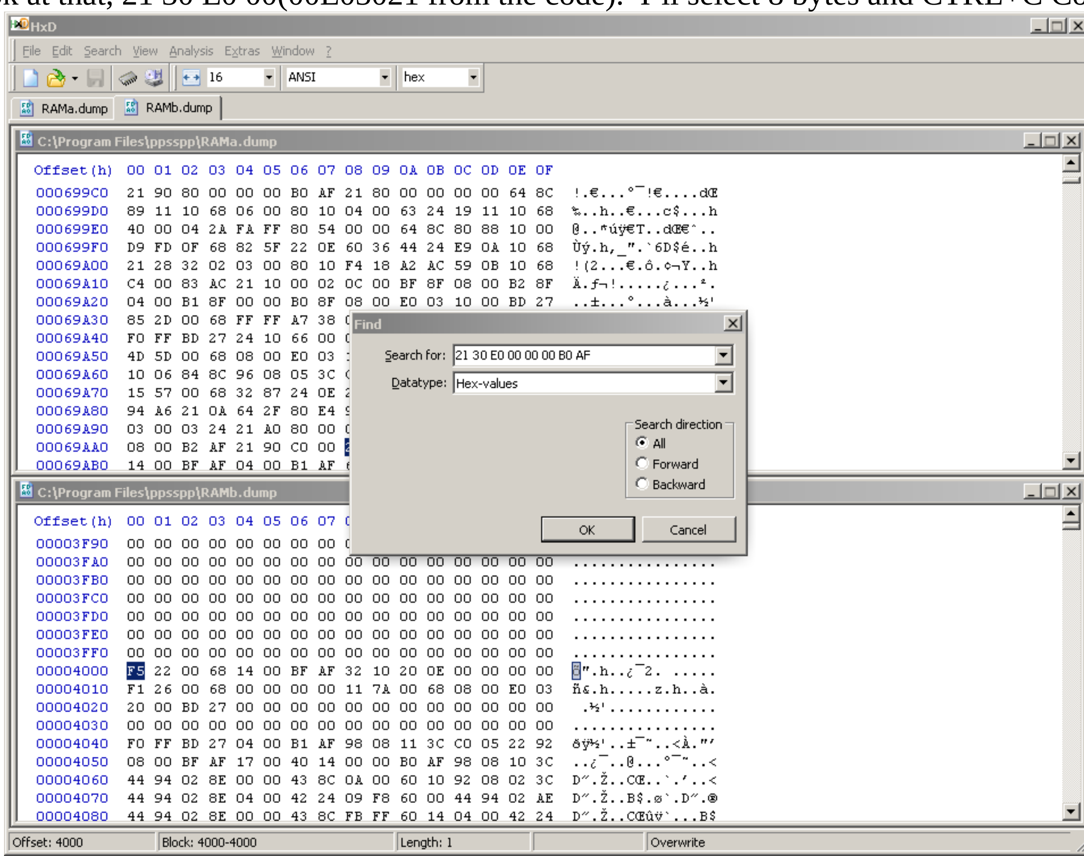
Then load those up for comparison in HxD.



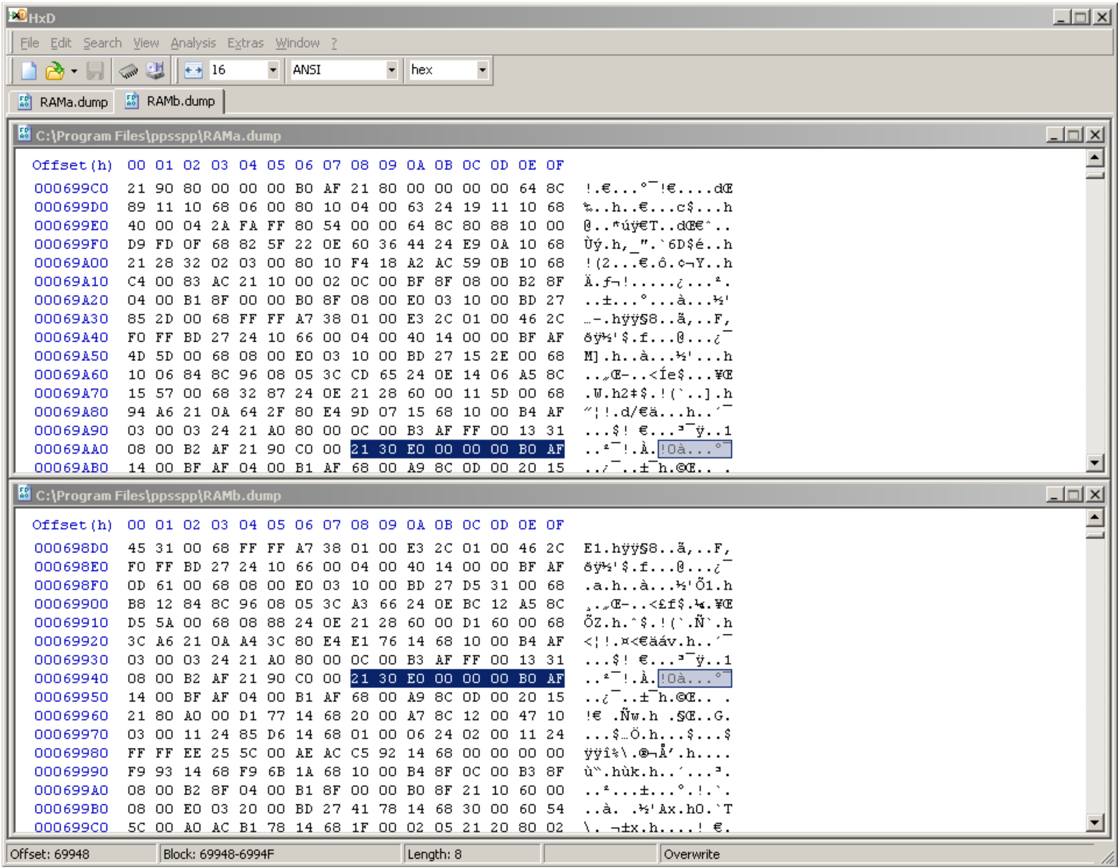
CTRL+G in the top window and select offset 69AA8



Look at that, 21 30 E0 00(00E03021 from the code). I'll select 8 bytes and CTRL+C Copy.



CTRL+F in the bottom window and CTRL+V paste, Datatype "Hex Values", Search direction "All".

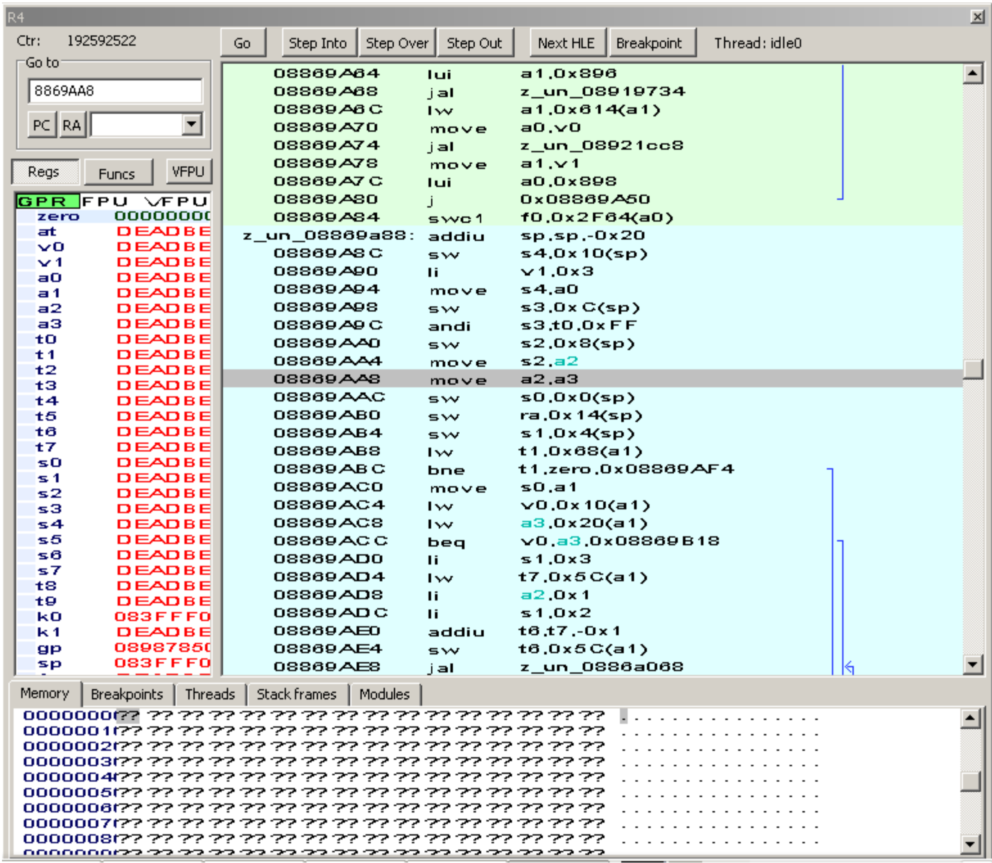


We don't just have a match but an exact match in RAMb.dump at offset 69948.

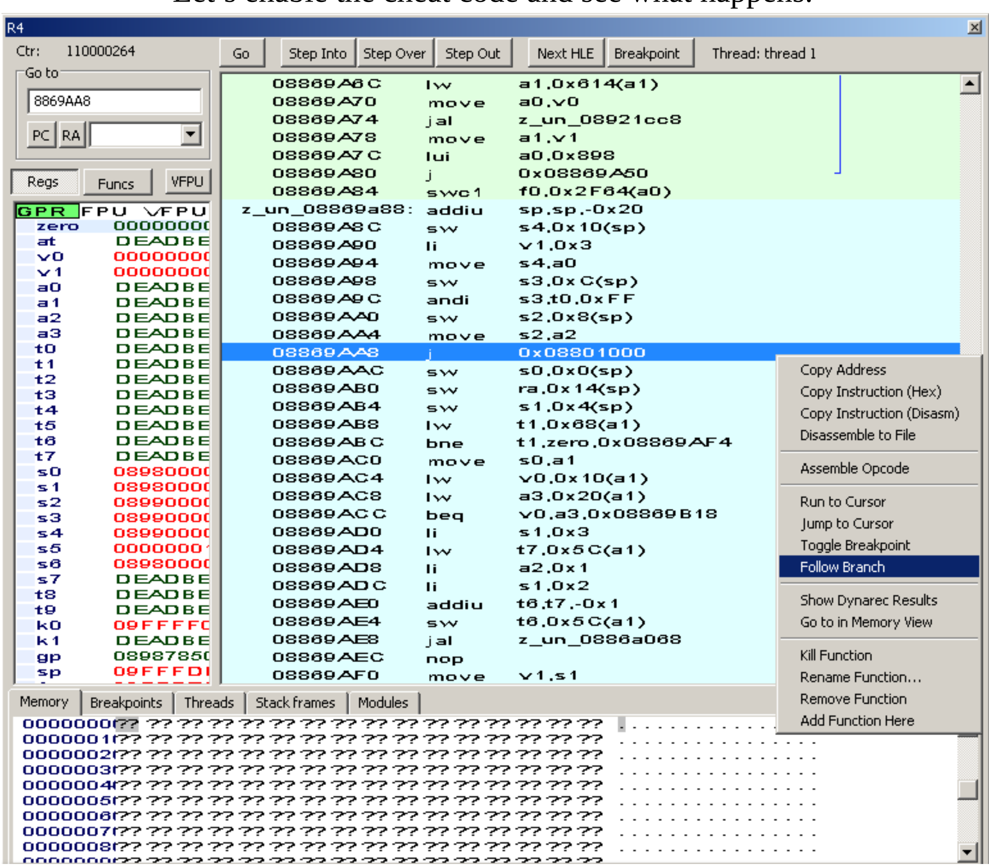
So our ported code begins to look like this:

_C0 Move x2 Hold Circle Norm
_L 0x20069948 0x0A200400
_L 0x20001000 0x00E03021
_L 0x20001004 0x0A21A6AC
_L 0x20001008 0x00129042

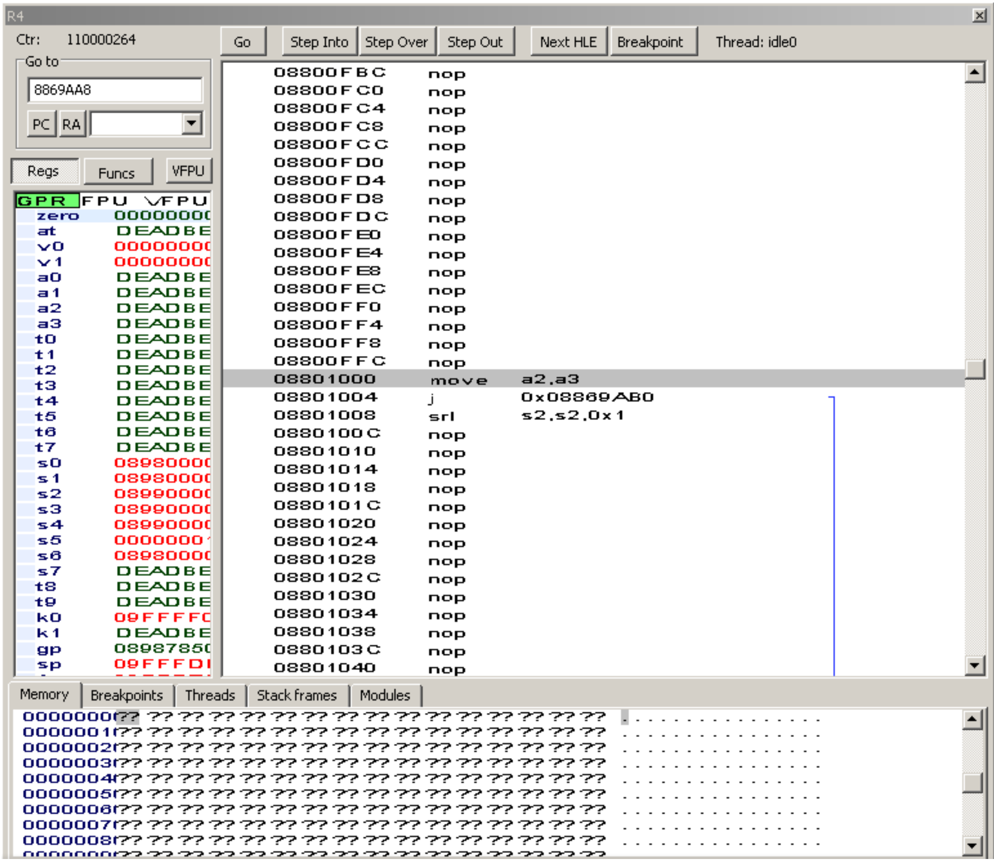
The problem is that portion in red, the return jump. We have to figure that part out. To do so, let's open up ULJS-00016 in the PPSSPP disassembler and look at its first code line address of 0069AA8 (08869AA8).



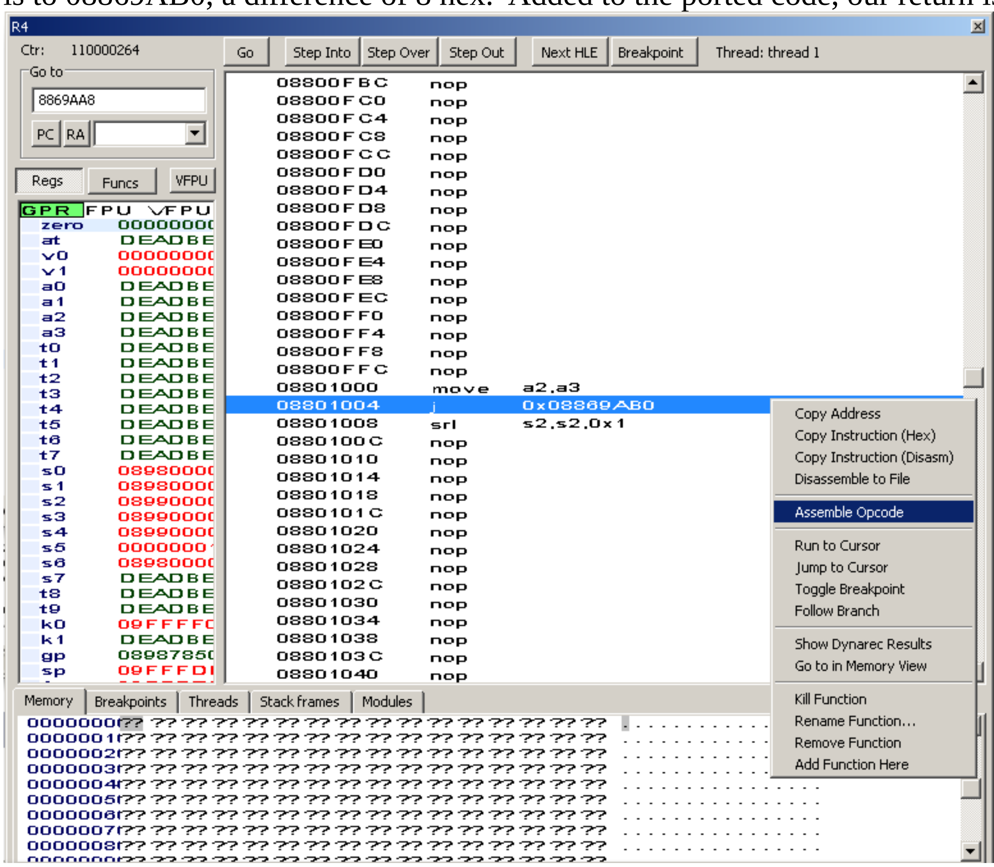
Let's enable the cheat code and see what happens.



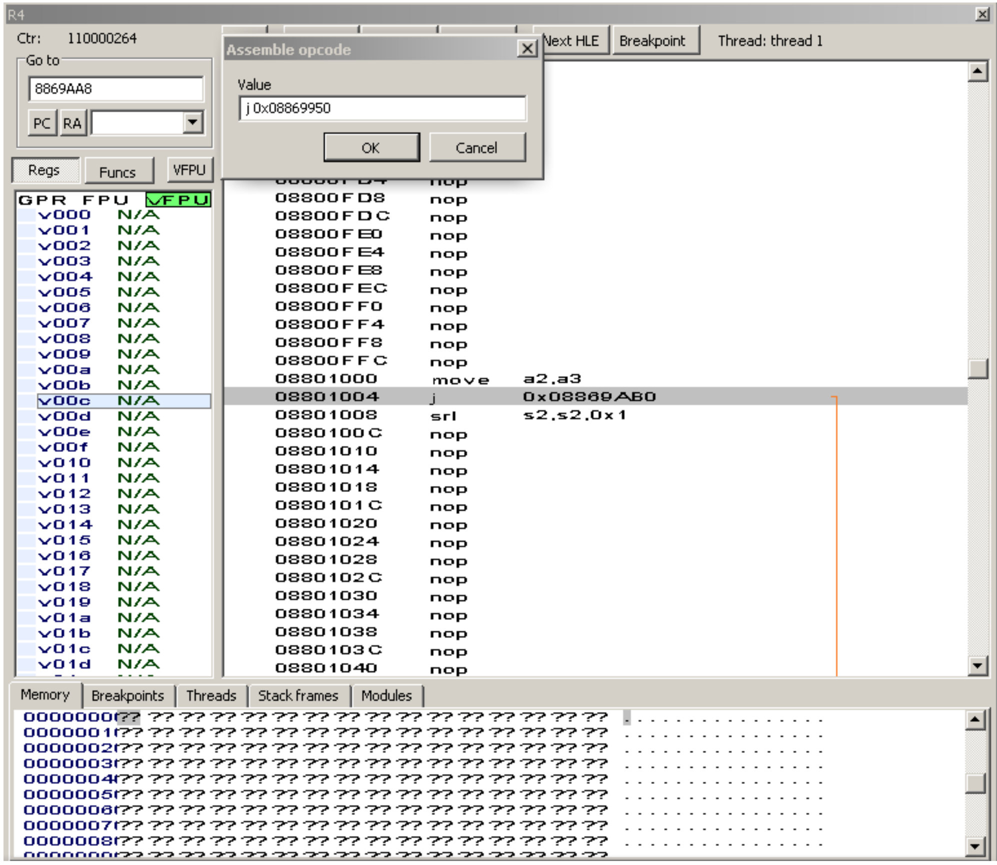
Now we can see the jump to 08801000. Let's follow it.



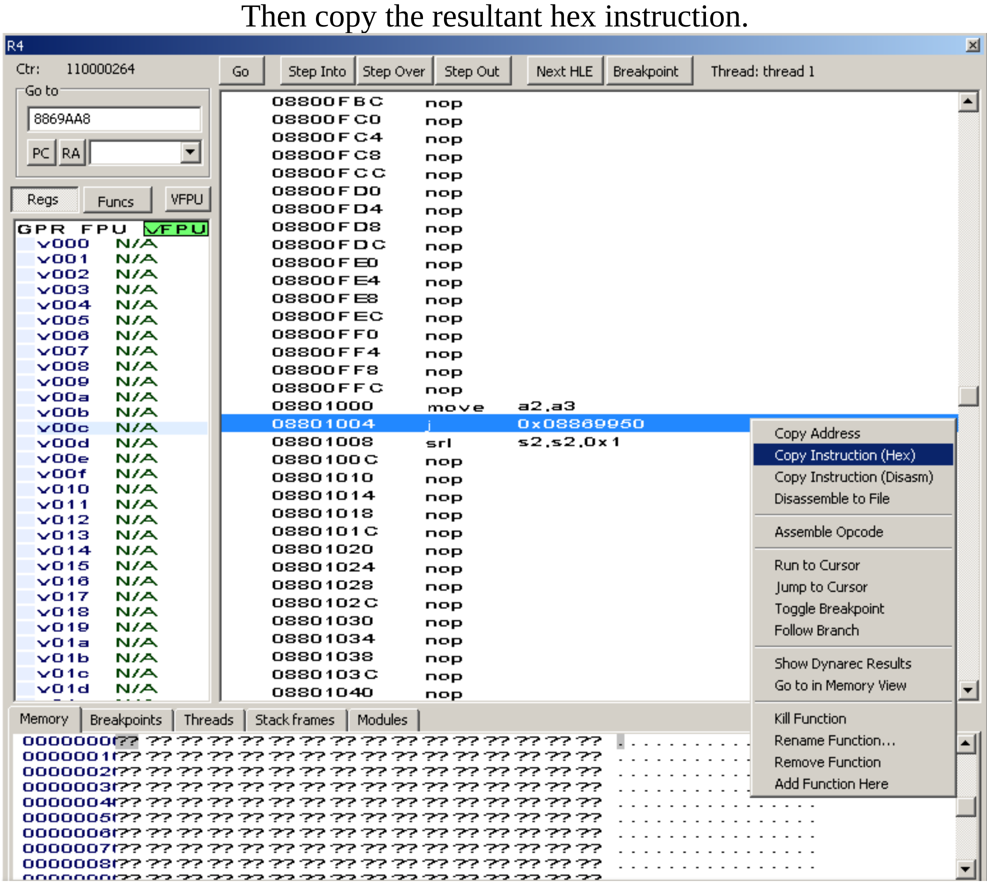
Return jump is to 08869AB0, a difference of 8 hex. Added to the ported code, our return is 08869950.



With PPSSPP paused, lets assemble an opcode “j 0x08869950”.



Then copy the resultant hex instruction.



Now I'll paste that hex instruction we just copied, it's "0A21A654". Let's add it to our cheat code:

_S ULUS-10022
_C0 Move x2 Hold Circle Norm
_L 0x20069948 0x0A200400
_L 0x20001000 0x00E03021
_L 0x20001004 0x0A21A654
_L 0x20001008 0x00129042

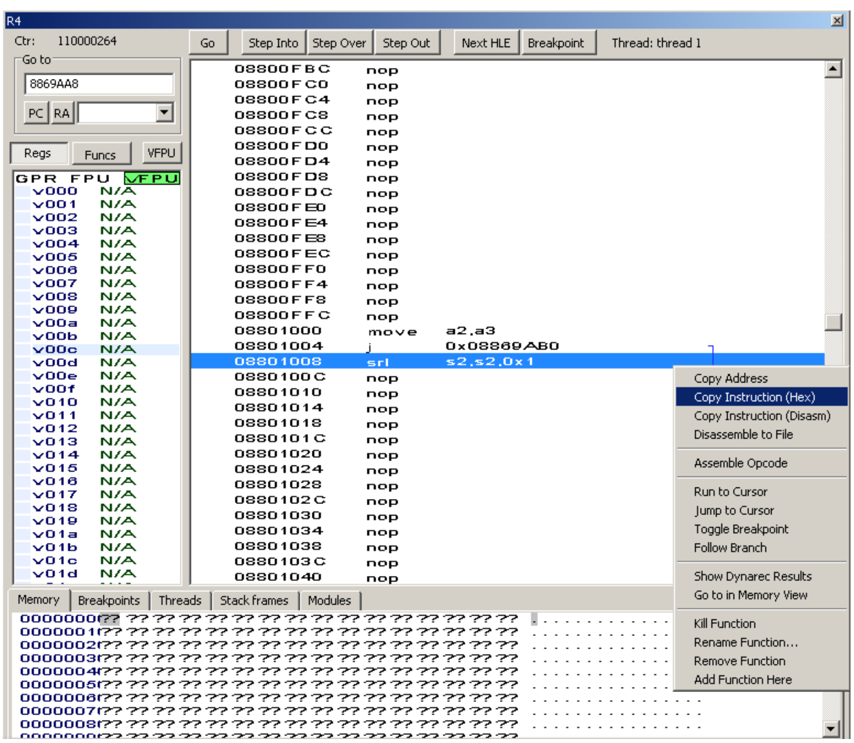
Compared to the original:

_S ULJS-00016
_C0 Move x2 Hold Circle Norm
_L 0x20069AA8 0x0A200400
_L 0x20001000 0x00E03021
_L 0x20001004 0x0A21A6AC
_L 0x20001008 0x00129042

So that's our ported code with amended return jump. The question this all brings to mind is, if both games are the same except for language, why does the code need to be ported?

Language is embedded in the primary executable. Our cheat codes are based on absolutes. This is exactly where, by memory address, we're going to make a change. It assumes the code-base isn't just similar but identical.

Doesn't take much to change the code-base. Just a singe function, or a few lines of text, could push all of those addresses around. That's why the code must be ported. If the game engine was identical and used variable translation tables for all text, like is done with the PSVita, cheat codes would work on every version of the game and regionalization would be a cinch.



By the way, if you're curious about that last line of code, "_L 0x20001008 0x00129042"

We'll just copy the hex instruction and paste it here, "00129042".

And that about sums up this tutorial. It's geared toward an advanced user with emphasis on code manipulation, not creation.

Our code port example was also a best case scenario, 100% match. Don't expect it to be so easy. Then again, it was a complex cheat code that altered program flow and inserted new instructions for the MIPS processor to handle. Many examples of code ports don't do that.

The goal of this was simply to familiarize the user with data-structures and tools. Individuals, based on their skills, will further refine techniques to make the best use of this information.

Conclusion

I'll conclude by saying what every good cheat coder will tell you. Don't use the cheat engine to do something that should be handled in MIPS assembly. Using two examples found in this text:

```
_C0 Avin 999 HP/MP
_L 0x802EECBA 0x00040001
_L 0x100003E7 0x00000000

_C0 Item All A
_L 0x408077B0 0x00070001
_L 0x0A0A0A0A 0x00000000
_L 0x208077CC 0x010A0A0A
```

The "999 HP/MP" is a bad code because it's fighting to set a value that is very dynamically altered by the program. The answer isn't to fight the flow of water with another water source, but to turn it off. Use a breakpoint to trace the code that decrements these values and stop the function at its source.

The "Item All" is acceptable because it makes efficient use of the cheat engine to write a primarily static value/address. It wouldn't make sense to do this in MIPS assembly through alteration of the program flow.

Keep in mind that, even for MIPS reprogramming, User Memory is fairly limited so there's a finite number of things you can redirect.

Thank you for your attention
noabody