Loot for SNES

Matthew Schneider

May 17, 2023

For my final project I implemented the compiler for Loot targeted for the Super Nintendo Entertainment System (SNES). The target language of the compiler is 65816 assembly. I've included instructions to compile and test programs as READMEs in the `loot/` directory.

1 The idea of the compiler

The AST and the parser remain roughly the same as the lecture implementation with the only differences coming with actual differences in the language features. The compiler outputs assembly code for the 16-bit 65816 CPU that the SNES uses. I have modified the compilation of each feature to work similarly to the a86 compiler as much as possible. But, of course, it will be different in a way that reflects the limited architecture: 16-bit operations, three registers (one accumulator register and two index registers), and only one-operand instructions. The largest changes are documented here.

In a86 we used `rax` to store the result of computation. There is only one reasonable choice for the 65816, which is to put the result in the accumulator A since it is the only register that supports all of the main operations of the CPU. The two other index registers X and Y is used often for scratch memory and storage. The heap pointer, which lived in `rbx` lives in main memory, since there aren't enough registers to freely give one up for this.

I have tried to avoid making the compiler directly use main memory as temporary storage or other scratch memory. Instead, temporary storage is done by moving data around the registers or using the stack. The only time that main memory does get used is for the heap pointer, indirect jumps, and dereferencing pointers (the process of dereferencing is described later).

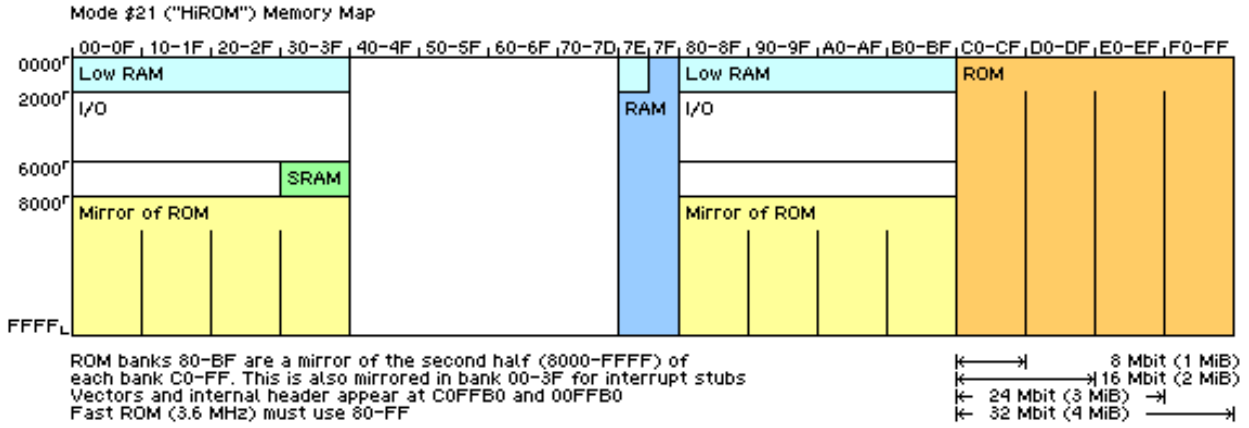
Here's a good reference for the instructions of the 65816 architecture: `http://www.6502.org/tutorials/65c816opcodes.html`. And here's a reference for the entire SNES hardware: `https://problemkaputt.de/fullsnes.htm` (probably not as useful).

2 Memory layout

2.1 CPU Memory Map

The 65816 CPU features a 24-bit address space (for up to 16 MB addressable memory), which can be thought of as 256 64K banks (where the highest byte of an address is the *bank byte*). This disparity between 24-bit addresses and 16-bit words becomes a major point of contention and a big pain in the ass. I have tried to avoid it as much as possible by forcing local address ranges to fit within 16-bits (for example ROM size as you will see).

The SNES maps different components of the console (e.g. RAM, ROM, hardware registers, etc.) to this address space. All compiled code uses the *hirom* mapping mode, which looks like:



Important bits include: ROM, from addresses `$C0:0000` to `$FF:FFFF` (note: `$` is the prefix for hexadecimal in the 6502 assembly family), which is also mirrored to other banks; RAM, from addresses `$7E:0000` to `$7F:FFFF` (addresses `$7E:0000` to `$7E:1FFF` are mirrored to other banks); and hardware registers, which live at various particular addresses in bank 0.

2.2 Layout of ROM

The result of compiling a program is a ROM of size 64K bytes (one bank). We only need 16-bits to address anywhere in the ROM (useful for function calls). Here is how it is laid out:

Addresses	Contents
`$C0:0000 - $C0:7FFF`	**Compiled code**
`$C0:8000 - $C0:FFAF`	CPU interrupt handlers SNES initialization routines Runtime system Graphics/tilemap data
`$C0:FFB0 - $C0:FFFF`	SNES ROM header 65816 interrupt vectors

The location of the ROM header and the 65816 interrupt headers are hardcoded (the former by Nintendo and the latter by the architecture). With this layout, the maximum size of compiled code is 32 KB. Any space not used is padded with 0s.

2.3 Layout of RAM

Here is how stuff in RAM is laid out:

Addresses	Contents
`$7E:0000 - $7E:00FF`	**Zero page** scratch RAM RTS global variables heap pointer other utilities
`$7E:0100 - $7E:1FFF`	**stack area**
`$7E:2000 - $7E:7FFF`	RTS reserved (mostly unused)
`$7E:8000 - $7E:FFFF`	**heap area**
`$7F:0000 - $7F:7FFF`	unused

Note that `$7E:0000` to `$7E:1FFF` is mirrored to bank 0. This means we can think of that area the same as `$00:0000` to `$00:1FFF`. This is actually necessary for the stack (described later) and allows for a few optimizations when using this memory directly.

3 The type system

On a86 there is the 3-bit split between immediate and pointer values. It's nice there because it makes sure all pointers are 8-byte aligned and we even have extra space to tag pointers. In 65816 with 2-byte alignment, so we can only freely sacrifice one bit. If we gave up the three bits, then we have only 12-bit integers, for a range of $-2,048$ to $2,047$. This is not a lot of integers.

The way that I got around this limitation is to merge all pointer types into one pointer tag, and store the actual underlying type on the heap with the data itself. The bit layout of pattern is as follows:

Type	ends in
Integer	`0`
Pointers	`01`
Characters	`011`
True	`01111`
False	`11111`
Eof	`101111`
Void	`111111`
Empty	`1001111`

Pointer	first word in heap
Box	0
Cons	1
Vector	2
String	3
Procedure	4

This approach requires another word to be used on the heap to store the actual underlying type. And, any pointer will require one round of dereferencing to figure out which pointer type it is, which is slightly slower. But this gives 15-bit integers, for a nice range of $-16,384$ to $16,383$.

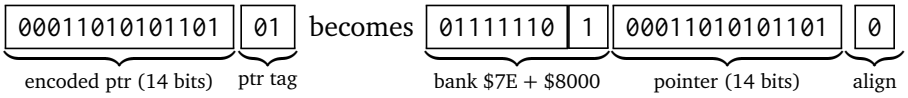
3.1 Characters

The character type in this language represents the ASCII range only (only 7 bits). Although, we have 13 bits available (and the SNES tilemap format could easily sneak in a 10-bit character type).

3.2 Heap effective addresses

For pointer types, we have only 14 bits to represent the 24-bit effective address (WRAM is only 128K big, so we actually only need 17 bits in order to talk about all of WRAM). To figure out the effective address of a pointer type, shift the bits in a word to the right by one, clear the least significant bit, and set the most significant bit. This gives us the lower 16 bits of our address. The bank byte will be `$7E` since that is the first bank that contains all of WRAM. This means that our pointer types can address memory from `$7E:8000` to `$7E:FFFF` (in 2-byte intervals) for a total of 32K of maximum heap space.

To recap, here is an example conversion of a box pointer and how we get the 24-bit address of the underlying data from the 14 bits that denote the pointer:



So the word `$1AB5` refers to the address `$7E:85DA`. For empty string and empty vector, we previously defined it setting all the pointer bits to 0 (a null pointer). However, in this scheme that would refer to the address `$7E:8000`, a perfectly valid address (the first one in the heap!). So instead I did the wasteful

thing by allocating heap space for every string and vector. This also changes the semantics of a program since `eq?` will not necessarily be true when comparing two empty string or vectors.

To actually load or store the data at some address in the heap, the effective address is first resolved using the above process. There is no offset mode like in a86 (so one cannot use the value in the main registers as a pointer). Instead, that address is then written directly to some predetermined area in memory, which is then written/read indirectly using the corresponding instruction.

3.3 types.rkt

In a code management improvement, the Racket `types.rkt` and C `types.h` header file that encodes information about the type system has been consolidated into one `types.asm` file. The assembler `asar` supports custom definitions and macros, so my compiler generates code with these definitions left intact for the assembler to decode. For example, the integer 42 would be compiled as `LDA.W 42<<!int_shift` (`!int_shift` is 1 as defined above). This avoids needing to maintain both files simultaneously, should the type system change (it did a lot in development).

4 The stack

There is no need to follow a human-imposed stack discipline (like System V) in 65816. However, the stack should never be left in an invalid state, since the CPU may be interrupted at any time (and the process of interrupting will use the stack).

The stack pointer in 65816 follows the empty-stack convention. This means the stack pointer is a pointer to the first address in memory that is *not* on the stack.

4.1 Limited stack size

The stack pointer is only 16-bits wide. In this case, the CPU treats the bank byte, as 0, and so the stack pointer dictates the lowest 16-bits of the address. The SNES itself only lays out RAM in bank 0 from addresses `$0000` to `$1FFF`, leaving us with only 8K of maximum stack area.

On top of that, I reserve the zero page (addresses `$00:0000` to `$00:00FF`) for the runtime system, scratch, and other important global variables like the heap pointer, since the CPU can address this specific region more quickly than others. (Unfortunately, the compiler does not actively take advantage of this speedy memory region.) Thus, the stack can be a maximum of just 7,936 bytes. There is no checking done as to whether or not the stack pointer touches this zero page area or even overflows.

4.2 Manipulating the stack pointer

At the end of functions we have to deallocate in bulk the area on the stack reserved for the parameters. However, the stack pointer in 65816 is *not* a general purpose register, so we can't just subtract from it like in a86. There are two approaches to shrink the stack:

1. Just put a bunch of pull instructions into a trash register until the stack is sufficiently shrunk, or
2. Move the stack pointer into the main registers so that it can be manipulated, then put it back.

The first approach is proportional to the number of things that need to be pulled (in this case, 5 CPU cycles per word). The routine I wrote for the second takes a constant 13 CPU cycles. So if fewer than three words needs to be pulled, then the compiler will just use multiple pull instructions. Otherwise, it will take the second option. I don't really know why I did this optimization; I was really just bored at the time I did this.

5 Functions

The calling convention used in Iniquity, which puts the return address below the arguments on the stack, means that we can't use the `call` instruction of a86, and likewise will prohibit the analogous instruction `JSR` in 65816.

Jumps can be done to either 16-bit or 24-bit pointers. Thus, we can just use 16-bit values for function pointers which plays nicely with our 2 byte word size (both in the stack and the heap). With a 16-bit pointer, the lower 16-bits of the program counter is changed and the bank byte does not change. A caveat of this is that all compiled code must live within the same bank, limiting us to maximum of 64K bytes of compiled code (the entire ROM lives inside bank `$C0`).

5.1 Indirect jumps

In this language a function application only checks for a valid procedure at runtime, which also means the jump address cannot be resolved until runtime. Instead of using a direct jump, where the target of the jump is known at compiletime (like `JMP $1234`, with effective address `$C0:1234`) we instead have to, at runtime, resolve the address of the jump, store it in RAM, then use an indirect jump (I used `JMP ($0000)`, where the lower 16-bits of the effective address is located at RAM address `$7E:0000`).

6 Printing

In lieu of the input/output features from Evildoer, I implemented the following print primitives: `print-int`, `print-char`, and `print-bool`. They work by first checking the type, then making a call to the runtime system. `print-int` will convert the number to a base 10 representation, and `print-bool` will print either `#t` or `#f`.

From these we can print any other value, using a function written in Racket. I wrote a `print-value` function in Racket that will accomplish that (which is in `print.rkt`), but I also implemented a few extra primitives to make things easier along the way:

- `cond`, with the AST and parser from Fraud-plus and an adaptation of my assignment code
- `integer?` and `boolean?`

7 Miscellaneous differences and optimizations

7.1 Software-interrupt error handling

In the a86 implementation, errors were invoked with the calling of an external error function defined in C. In the 65816 implementation, I use a software interrupt using the `BRK` instruction. Upon reaching a `BRK` instruction, the CPU will jump execution to an interrupt vector specified at some fixed address in the runtime. In this case, all the interrupt handler does is display `err` to the screen, make the screen red, then halt the CPU.

7.2 Evaluation order of binary operations

The binary operators that require integers (+, -, =, <) are compiled such that the behavior is different than on a86:

- a86: the arguments are evaluated left-to-right, and then types are checked

- 65816: the arguments are evaluated right-to-left, with the value of the right argument type checked before the left argument is evaluated.

Thus (supposing that `print-int` and `print-bool` were implemented in a86), the expression

```
(+ (print-int 97) (print-bool #f))
```

is an error in both languages, but a86 would print `97#f` before the error, and 65816 would print `#f`.

Why do I do this? Because of subtraction. The `SBC` instruction takes one operand, in this case, a value located on the stack, which is then subtracted from the accumulator. Therefore, we need to evaluate the second argument first so we can push it to the stack before evaluating the first argument. Type-checking is also done as soon as the argument expressions are evaluated because again, the we cannot as easily check the value while it lives on the stack.

For pure code, this won't change the result of a program. Overall, this compromise reduces the complexity of the generated assembly.

7.3 Block move optimizations

The 65816 does feature a set of block move instructions, `MVN` and `MVP`. The compiler uses these instructions in certain situations:

1. For string literals, instead of compiling a chain of loading each character then storing it on the heap, the string is instead stored as raw data (that actually lives in the middle of code skipped over by a branch). The raw data is then moved from ROM to the heap using the block move instruction.

2. For tail call optimization, the `move-args` function in a86 compiles a series of instructions for each value that needs to be moved down the stack. In 65816, I just block move it. Note that the source and destination of the move may overlap; this is why there are two block move instructions (`MVN` moves in increasing order of addresses, `MVP` in decreasing order). The appropriate one to use here is `MVP`.

8 Racket 65816 package

There is a basic package for representing the instructions of 65816 assembly in Racket (which can be found in `65816.rkt`). There is a struct for each instruction and each possible operand type, which can then be converted to a string representation which can be written to a file and assembled. There are also a few extra structs for particular assembler directives that are used (for example to denote data for string literals).

For example, the instruction `LDA #$53`, which loads the immediate `0x53` into the accumulator, is represented using this system as `(Lda (Imm #x53))`. However, not all of the instructions accept all the operand types in 65816; the malformed instruction `(Sta (Imm #x53))` (trying to store the accumulator with an immediate) will be accepted by the library. I don't check that a valid operand type is used, because the assembler will fail to assemble that program anyway.

9 The runtime system

The runtime system is a simple SNES program that will initialize the console, setting up hardware registers, clearing RAM, and calling the compiled code. It also handles displaying output to the screen and communicating with the SNES picture processing unit (PPU).

To display things to the screen, you need the graphics and a tilemap (which you can think of as a table of graphics). I drew a set of ASCII graphics, which are stored in VRAM (and sent over during initialization). However, VRAM lives on the PPU bus rather than the CPU bus, so the CPU can only write to it through a hardware register. VRAM can only be modified while the PPU is *not* drawing an image to the screen.

The tilemap is also stored in VRAM. The process by which the print primitives draws characters to the screen is described as follows:

1. All characters drawn by a print primitive are sent to a *copy* of the tilemap located in WRAM (which can be freely modified at any time).

2. During every VBlank period, essentially, at the end of every frame when the PPU is finished drawing the image, there is some time where VRAM can be modified. The tilemap buffer in WRAM is then sent over to VRAM using a DMA controller in the SNES.

3. The PPU draws the next frame, so the results of the most recent set of print primitives from the last frame are now visible.

10 Testing

I tested this compiler extensively, which also necessitated creating a testing framework. The test cases include all test cases from the lecture code that don't use I/O or pattern matching. It also includes some cases that I wrote, mostly from when I was implementing all the features.

10.1 The process of testing

Testing is done using a Lua script and the Mesen emulator. This allows tests to be run from the command line and the results to be printed to the terminal. For each test case, a ROM is created that contains both the test code and the expected output. The emulator will the test code and then print the value to the screen using the `print-value` function mentioned earlier, so that we have some kind of string representation of the result. To signal to the Lua script that the test code has finished running, the runtime writes to a predefined invalid location in memory (a no-op to the console but a trigger to the script).

Once the emulator receives the signal that the program is done, it reads directly from the CPU memory, extracting the output of the program as a Lua string. It also reads directly from the ROM to get the expected output also as a Lua string. These strings are then compared and that is what determines success or failure of a test. The testing script also handles the cases where the expected result is an error, or the program incorrectly errors during testing.

10.2 Drawbacks

Testing as strings can be flawed, since it's possible for two different programs to output the same string. A trivial example would be something like `(lambda (x) 42)` and `(lambda (x) 84)`, which would both print out as `#<procedure>` (although to see they are different you could apply both of them to something). It is also heavily reliant on the `print-value` function which gets compiled into each test ROM, so the testing framework is also implicitly testing this function and every language feature it uses for each test.

11 Conclusion

Can this be used to create your own SNES games? No. Unless your idea of a game is a really boring one that just displays text and doesn't take in user input at all. So what do we get out of it? I don't know.

It's possible that with a few extra features (inputs, display objects, graphic primitives, etc.) you could make an interactive program. However, the compiler has next to no optimization, and given the sheer speed of the SNES CPU (3.58 MHz), making sure that advanced logic can run quickly enough for a 60 FPS game will limit the overall complexity of any game. Additionally, it would require abstracting away the details of displaying images to the screen on the SNES in a way that would make it easier to use and compatible with other language features, which is hard.

Despite my pessimism, I did find the project very enriching. This was yet another deep dive of mine into the intricacies of the SNES, which has bequeathed unto me even more knowledge than I had known before. And, I even got to store local variables on the stack (which for strange historical reasons is not very idiomatic for SNES games).