Super FX Tutorial

- Made by DiscoMan -

A.K.A DiscoTheBat

Index

Disclaimer

By downloading this tutorial, you agree that you wish to work with the Super FX related assembly and/or hardware and that you will not hold the author, responsible in any way. Downloading this tutorial means that you have understood and agreed to all the terms and conditions of this disclaimer.

The author does not take any responsibility and is not liable for any damage caused through use of this document, be it indirect, special, incidental or consequential damages. The author agrees that this document can be reproduced, in whole or in part, under the circumstance that this document may not be edited nor hosted and/or used without the author being properly and clearly credited.

All informations presented in this tutorial are based on researches and author's own experiences.

This document isn't done for profit but merely for research and study purposes, giving the fact that such information isn't widely available on internet nor books.

Preface

I must start saying that this is a tutorial written by me, DiscoMan and my goal here is to teach how Super FX works, how to use it's assembly language options and how to successfully manage to do working codes and explain how plotting works.

I assure that this tutorial will settle, all doubts, questions, and even misconceptions about this chip, how it works and etcetera.

I'll squeeze my knowledge in this tutorial, though, since I can't teach anyone every time how things are supposed to work but I'll do my best here and I hope you may like it.

This won't use Super Mario World examples, as this tutorial just teaches how everything works, what's needed, what's similar of SNES instructions, it's advantages, disadvantages, etc. It's up to you to keep reading this tutorial and learn how this wonderful chip works as it's a very advanced, yet useful hardware piece that Argonaut Games developed for the equally awesome game console ever, the SNES.

I hope you may like it and remember, to learn ANYTHING, be sure to read carefully, do the examples, pay attention to even the smallest stuff and the most important of all, do it yourself, it's like math, you can't learn until you do it.

Chapter 1: Super FX Introduction

Welcome to the amazing world of Super FX! In this tutorial, you'll learn the story behind Super FX, it's ASM, some nifty features, it's pros and cons and a lot more!

To start, I'll talk about this chip, according to Wikipedia, this chip was developed by Argonaut Games thanks to Jeremy “Jez” San and Ben Cheese and co-produced by Nintendo to demonstrate the capabilities of polygon rendering in SNES, because of the technology used, Super FX, thanks to Star Fox, became the most successful RISC processor at the time, the idea of 3D games was so inspiring that the producers at Nintendo supported the idea to create a SNES with GSU built in, thus augmenting the power of SNES itself, however, the idea was scrapped due to costs and time.

This chip is a RISC graphic unit or GSU (Graphics Support Unit), which means that you can work with graphical features without slowdown, and not only that, but it can be used for other routines to make it faster, you can't simply use that as a graphical unit as you can freely program your own code to make it be run by this chip. This chip is a useful general use processor and it's up to the programmer to do anything he or she wants as long both GSU and SNES can read/write the resulting code, after all, GSU still needs SNES to interpret it's data.

Talking about speed, there are 4 versions of Super FX, one is the MARIO chip (or Mathematical, Argonaut, Rotation & I/O) and it's the first version of GSU, then being followed by it's namesake Super FX and at the time, both CPUs was clocked at 21 MHz internally but behaving as a 10.5 MHz CPU, then Super FX 2 version was launched, featuring it's full 21 MHz clock speed together with the last version which was a Service Pack containing fixes and optimizations.

The chip was used in very few games thanks to the end of the fourth generation of game history, making this chip rather underrated but even with very few titles, it has it's history, two games that used Super FX was in the best-selling games of SNES at the time with Star Fox and Super Mario World 2: Yoshi's Island. This chip went quite obscure thanks to it's hard ASM and some limitations but as they say, good things comes in small packages!

Now, technically speaking, Super FX ROMs can't have more than 2MB (or 1MB if GSU-1) in size which makes pretty difficult to do a bigger work without getting near the size limit, although GSU ASM uses a fewer commands, thanks to it's RISC design, also, GSU can't access the ROM/RAM at the time, thus making NMI/IRQ being remapped to a RAM area so the game can access without reading crashing opcodes, some vectors need to be remapped too, also, it comes with bitmap emulation, fast arithmetic operations, fast CPU speed, cache, graphical support and pipelined instructions, just to reach the maximum speed Super FX can offer.

So, with all those stuff, is Super FX hard to work with it? Is it incompatible with my stuff? Will it bring good results to my project? With those questions in mind, which I'll answer in the FAQ, don't worry! This tutorial is here to help you, so enjoy!

Chapter 2: Technical Stuff, Pros and Cons

(𓀀𓀁𓀂 𓀃𓀄𓀅𓀆𓀇 𓀈𓀉𓀊𓀋)

As you may know, Super FX has a variety of technicality which makes everything harder to understand and work, the goal of this chapter is to show and answer about technical stuff in a way that even a non ASMer could understand it, as this tutorial is for anyone, not just for ASMers...

To start, let's talk about the hardware specification, we'll be dealing with the version “2” of Super FX but it doesn't matter what version of GSU you choose as their instructions (albeit speed) are different but for simplicity sake, GSU-2 will be used in this tutorial; GSU has a speed of 21 MHz clock, can't support more than 2MB (1MB if GSU-1), can't access the ROM/RAM at the same time as SNES, it has graphical features, pipelining, cache, RISC like instructions and can work parallel with SNES CPU, just to summarize a few features GSU have. Even though there are limitations, GSU can prove itself as a valuable CPU.

Registers:

GSU contains 14 general registers from R0 to R13, those registers are 16-bit and SNES can read/write on then check the table below:

Register Name	Super NES CPU Address	Special Functions
R0	$3000-1	Default Source/Destination Registers
R1	$3002-3	PLOT Instructions, X coordinate
R2	$3004-5	PLOT Instructions, Y coordinate
R3	$3006-7	-
R4	$3008-9	LMULT Instructions, lower 16 bits
R5	$300A-B	-
R6	$300C-D	FMULT and LMULT instructions
R7	$300E-F	MERGE instruction, source 1
R8	$3010-1	MERGE instruction, source 2
R9	$3012-3	-
R10	$3014-5	-
R11	$3016-7	LINK instruction destination register
R12	$3018-9	LOOP instruction counter
R13	$301A-B	LOOP instruction branch

In this table, we can see that each register have specific functions in Super FX, this means that we can use those registers if we aren't using the functions as mentioned above, these registers even act the way we WOULD do on a SNES ASM, remembering that you can load AND store in those registers, as they're simply accumulators for any purpose.

The register R14 is the ROM address pointer, is still a 16 bit register and you use this register to load the address of data located in ROM, it's SNES code is $301C-D and SNES can read and write in this register, just remember that if any data is specified here (when GSU is operating), GSU will buffer the data pointed by this register, but other than that, it can be used

as a general register.

The register R15 is the program counter, it's a 16 bit register where GSU points the opcode data to be read, so if you modify R15, then you'll modify where GSU will read, this can be seen when you use any branch instruction as it'll modify R15 to effectively jump in the specified place, SNES can read and write and it's access data is from $301E-F

Register $3030-1 is the Status/Flag register, it indicates the status of GSU, SNES can access and modify it's bits to effectively determine GSU status, SNES can read and write on these too, it's like the processor status register on SNES, only 16 bit, in this table we'll show every bit contained in those addresses for SFR:

Flag	**Description**
IRQ	Interrupt flag
B	Set to 1 when the WITH instruction is executed
IH	Immediate higher 8-bit flag
IL	Immediate lower 8-bit flag
ALT2	Mode set-up flag for the next instruction (changes instructions)
ALT1	Mode set-up flag for the next instruction (changes instructions) (This and above is the $3031 Status portion, below is $3030 Flag Portion)
R	Set to 1 when reading ROM using R14 address
G	Go flag (set to 1 when GSU is running)
OV	Overflow flag
S	Sign flag
CY	Carry flag
Z	Zero flag

Next register is Program Bank Register (or PBR), it's an 8-bit register and it simply tracks the current bank of the currently executed opcode, SNES can modify and read this register as well and it's address is $3034

Next register is ROM Bank Register (or ROMBR), this is an 8-bit register and specify what bank GSU should read on ROM, when buffering operations are done, only GSU can modify and it can only be read on SNES, it's address is $3036.

Our next register is RAM Bank Register (or RAMBR), this is a 1-bit register and specify what bank GSU should read/write on RAM, the address is $303C and only SNES can read it, while GSU modifies it, it should be noted that you can choose banks $70 or $71.

The next register is the Cache Base Register (CBR), it specifies the starting address when data is loaded from ROM or RAM to the cache RAM, it's address is $303E-F, it's a 12-bit register and SNES can only read while GSU can modify it.

The next register is the Screen Base Register (SCBR) and this is used to specify the start address of character data storage area, this is used to point where data is located and/or to be stored, this address is 8-bit and SNES can write only, GSU has no access from this register, and it's address is $3038.

The next register is the Screen Mode Register (SCMR), this address controls the screen height during PLOT processing and it can control ROM and RAM bus assignments, also it controls the color (BPP) mode:

SCREEN HEIGHT

HT1	HT2	Mode
0	0	128 (pixels)
0	1	160 (pixels)
1	0	192 (pixels)
1	1	OBJ mode

COLOR MODE

MOD1	MOD2	Mode
0	0	4-color mode
0	1	16-color mode
1	0	NOT USED
1	1	256-color mode

This address controls how GSU plots graphics and how it open access to SNES bus when accesses/modifies ROM and RAM, so beware, it's a 6-bit address and SNES can only write to it, it's address is $303A.

Next register is the Color Register (COLR), this register specify the colors to be plotted when PLOT is executed, SNES can't mess with it in any way but GSU can work with it when COLOR and/or GETC is executed, this address is 8-bit.

Next address is Plot Option Register (POR) and it's the main way to change how PLOT instruction will work, it's a 5-bit address that SNES can't modify, only GSU with the CMODE instruction, how it works will be explained later.

Next register is Back-up RAM Register (BRAMR) it's a register that makes data “protected” after GSU saves data to Backup RAM (currently unused), it's 1-bit and SNES can write to it whilst GSU not, it's address is $3033.

Next is Version Code Register (VCR), it's an 8-bit register whose purpose is to read GSU version code, it can be only read by SNES and it's address is $303B.

The next address is the Config Register (CFGR) and it controls the speed of MULT instructions and sets up a mask for interrupt (IRQ) signal. SNES can only write to it. If Clock Select Register is set to “1” (by being 1-bit), then the MS0 (Multiplier Speed Selection) bit should be zero and vice-versa. This address is 8-bit and it's location is $3037.

The last register is a 1-bit address called Clock Select Register (CLSR) and SNES can write to it, it can simply controls GSU's clock speed, either 10.7 or 21.4MHz and it's located at $3039.

Pipeline:

This was the address list, now onto more technical details of GSU, next thing to talk about is the Pipeline.

Pipeline is a way to reduce the wait time of the processor by loading the next instruction while the CPU is executing the previous instruction, it works like this:

```
BNE Doop
ADD R0
[…]
Doop:
INC R5
```

In this code, BNE is executed and jumps to the label Doop, although it doesn't matter if the branch is executed but ADD R0 will be always executed as the next instruction, use that as your advantage. Also, beware of two byte instructions as it would load bad data, especially those who modify R15, such this for example:

```
BEQ Chop
BRA Chap
[…]
Chop:
DEC R1
```

The branch instruction is loaded but the next instruction instead of being loaded BRA Chap, the pipeline system only recognizes the BRA part and the DEC R1 as BRA jump address, as the Chap is excluded, so the code wouldn't work, you could overcome this situation by sandwiching needed instructions or a simple NOP if it's the case like this for example:

```
BEQ Chop
NOP
BRA Chap
[…]
Chop:
DEC R1
```

Think of pipeline as a way to double the speed when needed, as the CPU will prepare and load the next instruction instead of waiting one to be completed.

Cache RAM:

Next technical topic is the Cache RAM, C-RAM is a 512 byte high-speed memory, it's main function is to act as a way to reduce cycle waste and reduce processing time, also, it can be used to make GSU free the SNES CPU to access ROM or RAM, thus making possible to reduce the time SNES wait for GSU to finish it's business. If you couldn't understand this part, think that the data is located at ROM, because GSU will read it, it would interrupt SNES

but by loading the same code to C-RAM, GSU won't need to call ROM, so it'll just read directly into his own C-RAM, thus reducing overhead and because this RAM is fast, the completion of routine would be a lot more faster, I'll explain with more detail later.

Mappings:

GSU has a different mapping mode that unfortunately holds the ROM to a 2MB (1MB if using GSU-1) limit, it can't support either FastROM or HiROM albeit strangely enough, SNES can access GSU through this areas completely, even the manual states that banks $80-BF could be used (unused feature or just mirroring?), so probably GSU would have a different mapping support if it wasn't discontinued so quickly.

Anyway, ROM for GSU is mapped to 2MB (1MB if GSU-1), starting from $00:8000. More 2MB are used for ROM image and the image is stored in blocks of 32Kbytes.

The RAM for GSU is mapped 128Kbytes starting from $70:0000, it's basically the same as SRAM but considered an optional RAM for use on SNES.

GSU contains a possible Backup RAM located at $78:0000, it was never used so one can wonder if such thing would act as a SRAM while the “Game Pak RAM” would act as a BW-RAM, like SA-1.

Interruption:

As it's known, GSU can't access the ROM/RAM the same time as SNES and vice-versa and if this happens, SNES may crash because of wrong opcodes and the most probable time this may happen, is when SNES is reading an interrupt, so it's necessary to move IRQ/NMI to WRAM as some other vectors too, read the table below:

Interrupt Vector Address	Dummy Vector
00:FFE4	00:0104
00:FFE6	00:0100
00:FFE8	00:0100
00:FFEA	00:0108
00:FFEE	00:010C

Remember that if RON (GSU ROM Access) and GO (GSU is running) flag is 1 and SNES access the ROM data, it'll read dummy data by following the lower 4 bits of that address, read the table below:

Lower 4 Bits of Address	Dummy Data
$0,$2,$6,$8,$C	$00
$4	$04
$A	$08
$E	$0C
Other	$01

Remember this when hacking or creating homebrews using GSU as a coprocessor.

The GSU interrupt to SNES is different, when STOP instructions is executed, GSU will generate an IRQ to the SNES CPU, this means that if GSU ends it's operation, SNES can resume automatically, but when IRQ is generated for other reasons, SNES CPU must determine if GSU was the source of the IRQ. There's a IRQ flag at bit 15 of GSU status register. If the flag is “1”, then IRQ was generated by the completion of GSU processing. When bit 15 is read, then the bit is reset to “0”. The IRQ output by GSU can be disabled if you set the bit 7 of the CONFIG register by setting it to “1”.

Pros and Cons

I can't lie, everything has it's pros and/or cons and this isn't an exception, I'll did a simple list about what this GSU can do or can't, this part is purely optional, you can skip if you want:

PROS:

- Faster CPU speed (about 21.47 MHz)
- Graphical support
- Parallel processing with SNES CPU
- RISC Like instructions (this means that CPU design use less bytes and/or cycles to do the same thing SNES CPU would do it and it's even faster on a Cache area)
- Cache RAM (512 bytes)
- Pipeline processing
- It's possible to convert SNES opcodes to GSU opcodes with a bit of effort
- GSU is a general type processor, you can use for ANYTHING other than graphics if you desire, it's up to you to program anything at your heart's content
- Extreme overclock (60 MHz clock reported)

CONS:

- ASM requires some practice
- Doesn't allow simultaneous access to ROM and/or RAM (allows parallel access and you can transfer routines to WRAM for processing)
- ROM is limited to 2 MB and it's mappings use SlowROM
- Missing key instructions and features such as stack, for example

Chapter 3: Load, Store, Routine and Bank Stuff

Amazing! You made it to the start of GSU ASM knowledge, in this chapter, you'll learn how to Load, Store, Call banks and data on those banks and how to handle a subroutine.

As you know, GSU have 14 general registers for you to use as you wish, this obviously makes things easier for you to create routines as data would be saved on more than 3 registers, unlike SNES, since both registers are 16-bit, you can hold up a value until $FFFF or use them as an 8-bit register, again, unlike SNES, there's no need to specify the CPU to call 8/16-bit modes since GSU does that automatically for you, also, as a register, you load a value, store it, do some maths with it as you wish.

Since there's various way to work with registers, in this tutorial, we'll use any register as needed, so don't worry.

I'll show you, two simple examples of load and store immediate values operation, one using 8-bit data and another using 16-bit:

```
IBT R0,#$03
;This loads an 8-bit immediate into register R0
;IBT grabs the bit 7 of immediate and store from bit 8 through 15 of R0

IWT R1,#$7541
;This loads a 16-bit immediate into register R1
;IWT acts like IBT, except it's 16-bit, so if you want 8-bit values, IBT is the best choice unless you don't want sign extend

STOP
;Stops CPU from processing
;This acts like a STP, it is advised to put this code at the end of your routine
```

Unlike SNES, the load and store instructions are done in a single instruction rather than two, the memory cost for this operation is slightly reduced and there are less cycles used rather than SNES.

And, this is simpler than SNES, you load #$03 into R0 and #$7541 into R1, this data won't be cleared after doing this operation, so you could use to a later operation if needed.

Unfortunately, there's no way to effectively zero a register, but there are some tricks you could use to zero a register:

```
IBT R0,#$00
;Zero the register R0
;IBT grabs the bit 7 of immediate and store from bit 8 through 15 of R0
;Or, if some register is #$00, then you could use MOVE

MOVE R1,R0
;Load the contents of R0 and store into R1
;You could use MOVES to load flags with the operation

SUB R0
;Do R0=R0-R0
```

;To use with other registers, remember to use WITH

But, IBT/IWT only loads immediate data only, what if I happen to need to load/store data from ROM or RAM?

If the bank is already set, you can use 9 instructions, LDB (Rm), LDW (Rm), STB (Rm), STW (Rm), LM (xx), LMS (kk), SM (xx), SMS (kk) and SBK, like SNES, you need to assign the address on the register, so the CPU could load data, here's some examples of how to load/store data from ROM/RAM:

IWT R1,#$3482
;This loads a 16-bit immediate into register R0
;This points to a SRAM area.

LDB (R1)
;This loads the pointer at R1, grabs the byte and stores at the destination register which is R0
;70:3482 is $51, so R0 is $0051

STB (R1)
;This loads the byte data at source register R0 and stores again at 70:3482
;You must remember that you need to set the source and destination registers for to fully use this function, otherwise it'll use R0 as source/destination only

IWT R2,#$8000
;This loads a 16-bit immediate into register R0
;This points to a SRAM area.

LDW (R2)
;This loads the pointer at R2, grabs the word and stores at the destination register which is R0
;70:8000 is $FF and 70:8001 is $01, so R0 is $01FF

STW (R2)
;This loads the word data at source register R0 and stores again at 70:8000 and 70:8001

IWT R14,#$C000
;This loads a 16-bit immediate into register R0
;This points to a ROM Bank area, remember that R14 is used with GETB instruction!
;If you want to change a ROM's bank, make sure to use a ROMB instruction!

GETB
;This loads data from register R14 which pointers to the current ROM bank data which now is 00:C000
;Data from 00:C000 is #$9A, and transfer that to destination R0
;Byte is loaded to low byte register of R0 and high byte gets reseted
;Data is now #$009A

STB (R2)
;This loads the byte data at source register R0 and stores again at 70:8000

LM R0,($5678)
;This loads word data from RAM $5678 and $5679 to R0

SM ($5678),R0
;This stores word data from R0 to RAM $5678 and $5679
;Observation, if LM or LMS is used once, instead of using SM or SMS, you could use SBK instead.

;LMS and SMS uses short addressing, this means, only even numbers from 0~510 (I put the value in decimal)

Now, we're going to learn how to do subroutines, you may ask, doesn't GSU have a way to do subroutines? Yes and no.

While SNES can do jumps and after a RTS/RTL, restore the previous routine, GSU can't do that automatically, mainly because it lacks stack, you need to store the address of the continue routine any register you want (or use LINK, but LINK uses R11 as destination for it's calculations) and then you jump to the subroutine by either using JMP, LJMP or IWT, here's the example:

```
IWT R11,#ReturnLabel
;Load the ReturnLabel to R11

IWT R15,#SubroutineLabel
;Load the SubroutineLabel to R15
;Because R15 is Program Counter, then jump effectively to this label as specified

NOP
;Dummy

ReturnLabel:
INC R0
;Increase data in R0

[…]
SubroutineLabel:
ASR
;Shift data to right

MOVE R15,R11
;Move the ReturnLabel stored on R11 to program counter R15, thus returning to this label. By using IWT, R11 isn't always needed

NOP
;Dummy
```

As you can see, you don't have RTS/RTL, so you need to store the return to any register and after you're done with subroutine, then you move what's on that register to R15, basically you do a manual RTS. Remember that if LINK is used, R11 must be used instead to return.

But there's a way to save some cycles and space if subroutine return is closer to the jump, you can use LINK instruction like this:

```
LINK #4
;This instruction sums R15 with the number specified here and stores on R11
;It goes from 1 to 4

IWT R15,#SubroutineLabel
;Load the SubroutineLabel to R15
;Because R15 is Program Counter, then jump effectively to this label as specified
```

```
NOP
;Dummy

ReturnLabel:
INC R0
;Increase data in R0
```

As you can see here, LINK acts like the IWT that specified the return address, except this one acts faster and use less bytes, the only limitation is that LINK ranges from 1-4, while IWT is virtually unlimited.

Now, of course, no one would want to stay working with banks 70 or 00 only, correct? GSU has specific instructions for you to change banks, in those examples, you'll learn how to change both ROM and RAM banks:

```
IBT R0,#$08
;Set R0 to #$08

ROMB
;Load source register and set Bank to 08
;By now, every GETB instruction will load data from 08:XXXX

IBT R0,#$71
;Set R0 to #$71

RAMB
;Load source register and set Bank to 71
;Now data can be fetched from SRAM 71:XXXX
 ;RAMB is 1-bit value, you could also do IBT R0,#$01 too, #$71 seems an overkill but whatever
```

Chapter 4: Destination and Source

You can't do GSU ASM if you don't know how to set source and destination, this is a way to change what will be loaded or saved between registers or anything that changes register's data or exchange data to those registers, GSU has R0 as a default source and destination but even with that, sometimes you need to temporarily store or load values without need to doing that to SRAM every time and that's why Register Prefix Instructions are for, you change what you'll going to use or not, like this for example:

WITH Example:

```
IBT R5, #$FA      ;Load FA to R5
WITH R5           ;Set R5 as Source AND Destination
MULT R1           ;Multiply R1 with R5 and store the result to R5 (R0 becomes default again)
STW (R5)          ;Store whatever in R0 to address in R5
```

FROM Example:

```
IBT R5, #$FA      ;Load FA to R5
FROM R5           ;Set R5 as Source Only
MULT R1           ;Multiply R1 with R5 and store the result to R0 (R0 becomes default again)
STW (R5)          ;Store whatever in R0 to address in R5
```

TO Example:

```
IBT R5, #$FA      ;Load FA to R5
TO R5             ;Set R5 as Destination Only
MULT R1           ;Multiply R1 with R0 and store the result to R5 (R0 becomes default again)
STW (R5)          ;Store whatever in R0 to address in R5
```

As you can see, you get different effects for each opcode used, for example, WITH sets both source and destination apart from setting the B flag, FROM sets only source while TO sets destination. Beware of two things, when using WITH, since B flag get set, if the next instruction is TO, then MOVE instruction will be performed, if FROM is next, then MOVES will be performed.

And beware of all register prefix instructions though, even if you had it set once, you need to set it AGAIN, as every operation with the exception of TO, FROM, WITH, ALT and branches resets the destination and source to the default register, which is R0.

Chapter 5: Branching and Comparing

You can't learn basic ASM without learning how to limit code to conditionals when needed, since probably you want to do code to work if something is X or Y, branching and comparing is essential for starters in any ASM of any kind.

First of all, we'll start with how comparing works, since it's first needed so you could understand how branching works.

Before start, compare works quite different than SNES, you can't use immediate values, you need to input an immediate value in a register then use the compare command. The compare is done by SUBTRACTING the source register with register to be compared, then, it sets the flags accordingly, see this example:

```
WITH R1          ;R1 is source and destination
LDB (R2)         ;R2 loads a random address with #$30 to R1
FROM R1          ;R1 is Source
CMP R3           ;R3 have #$30; Subtract R1 with R3 which result is zero, thus, set the Zero flag and reset the other flags
BEQ Label        ;Is Zero flag set? If it is, then jump to Label
NOP              ;Dummy

[…]
Label:
STOP             ;Stop GSU
```

As you can see, different compare results sets different flags, for example, it can set O/V flag, S flag, CY flag and Z flag, it's just a matter of experimentation, here's another example, the same as above but with different result:

```
WITH R1          ;R1 is source and destination
LDB (R2)         ;R2 loads a random address with #$01 to R1
FROM R1          ;R1 is Source
CMP R3           ;R3 have #$02; Subtract R1 with R3 which result is #$FFFF, thus, set the Overflow flag and the negative flag but reset the other flags

BVC Label        ;Is Overflow flag clear? If it is, then jump to Label
NOP              ;Dummy

[…]
Label:
STOP             ;Stop GSU
```

You must know that the same applies for 16-bit values so be careful when you do a compare here.

There also other types of branching opcodes and they act different accordingly to the

condition it's using:

Minus-Plus branching
BMI branches if the operation uses a minus value (S flag is set).
BPL branches if the last operation isn't a minus value (S flag is unset).

Great or less and/or Carry flag
BCS branches if Rn = Equal or greater than the compared value. Additionally, this branches when the carry flag is set.
BCC branches if Rn = Less than the compared value. Additionally, this branches when the carry flag is clear.

Overflow and Sign flag
BGE branches if the overflow and sign flag is clear (Do Sign XOR Overflow = 0).
BLT branches if the overflow and sign flag is set (Do Sign XOR Overflow = 1).

Branch-always
BRA as stated, will always branch.

Another thing to notice is that JMP/LJMP are “branch” opcodes like those above, the difference is that flags are unset like any other instruction, and that you can use JMP for repeated addresses, without having to resort using IWT R15 all the time, saving time and space. The problem is that JMP is quite useless if you can use BRA if the range allows it.

However, LJMP is different since you CAN jump between banks so you may consider using LJMP if needed.

Branch-always special
JMP simply jumps as specified by registers R8-R13
LJMP unlike JMP, this one can jump between banks and put the address at the Cache Base Register, can be specified by registers R8-13 and Source Registers.

For LJMP, one must take care, the source register will act like the bank for one to set PBR (Program Bank Counter) while the value specified by Rn in LJMP will be transferred to R15, since there's no way to manually set PBR without resorting to SNES side operation, then LJMP is advisory and mandatory for any between work that goes between banks.

Chapter 6: Adding and Subtracting

GSU is a powerful processor with powerful mathematical capabilities, GSU have a dedicated ALU for mathematical operations such as shifts, bitwise operations and so on but it counts with operations for Adding and Subtracting, in this chapter we'll learn how to use those operations.

There are 7 instructions for adding and subtracting, apart from 2 instructions for increment and decrement and as a such, you, the user may want to use them, so in this chapter, we're going to learn how to use each instruction, to start, how to increment and decrement:

```
INC R0      ;Increment R0
STOP        ;Stop CPU

DEC R1      ;Decrement R0
STOP        ;Stop CPU
```

As you can see, there's no need to specify destination/source registers, unlike the next examples.

The next examples, deals with the addition and subtraction opcodes:

```
FROM R2     ;Set R2 as source – R2 is #$3D
TO R4       ;Set R4 as destination
ADC R3      ;R3 is #$01 and carry is set – R3+R2+CY->R4 = #$01+#$3D +CY= #$3F
WITH R1     ;Source and destination is R1 and R1 contains #$38
ADC #1      ;Do R1+n being n=1, also carry flag is unset = #$38+#$01+CY = #$39
```

These ADC instructions sums the values of source with the value in the register or the value as set in the #n instruction. Unlike ADD, this one sums the value with carry flag, so if CY is set, then the flag is added with the values specified, here more examples:

```
WITH R5     ;Source and destination is R5 – the value is #$4283
ADD R4      ;Add R4 with R5 – R4 is #$2438 = #$4283+#$2438 = R5 = #$66BB
WITH R2     ;Source and destination is R2 and R2 contains #$AA
ADD #7      ;Do R2+n being n=7 = #$AA+#$07 = #$B1
```

As noted, unlike ADC, this one doesn't use carry flag albeit it can set the CY flag under certain conditions, so it's useful if you doesn't want CY flag to mess around with values. As for subtraction instructions, the same thing happens, here some examples.

```
WITH R5     ;Source and destination is R5 – the value is #$5682
SBC R4      ;Subtract R4 with R5 – R4 is #$3609 = #$5682-#$3609-CY = R5 = #$2079
WITH R4     ;Source and destination is R8 – the value is #$753A
SUB R8      ;Subtract R4 with R8 – R4 is #$426B = #$753A-#$426B = R5 = #$30EF
WITH R2     ;Source and destination is R2 and R2 contains #$309B
SUB #7      ;Do R2+n being n=7 = #$309B-#$07 = #$3094
```

With this last piece of example, we finish this chapter by saying that GSU and SNES has a similar instruction and way of setting the adding and subtracting, just be careful that there's a fewer ways to clear the CY flag, like by clearing the flag using SNES to clear GSU's registers or wasting some bytes and use GSU for that, like using MOVES instructions for example, however, such instructions are useful so don't worry about them, just follow the examples and you'll be fine.

Chapter 7: Shifting and Multiplication Operations

GSU have powerful arithmetic instructions that may even surpass DSP-1, thanks to the way the chip was designed along with it's higher clock, independent of this fact, GSU mathematics instructions are similar of those in SNES, except that it can do 8 and 16 bit calculations and results can be either 8, 16 and 32 bit.

Apart from that, it can do bit-shifting operations, like ASR, LSR, ROR and ROL thanks to ALU. Continuing even further, GSU, when properly programmed can even perform DSP-1 like calculations, without further ado, we'll start with bit-shifting operations:

Consider that value in R5 is #$4F7B

```
WITH R5         ;Source and Destination are R5
ASR             ;Arithmetic shift right two times
WITH R5         ;Source and Destination are R5 (again)
ASR             ;Now the value will be #$13DE
FROM R5         ;R5 is Source
STW (R0)        ;Store whatever is in R5 to R0
```

Everything in bit 0 will be transferred to carry flag while bit 15 won't be affected in any way, take that in mind if you need to use the carry flag for something else and/or keep the MSB set.

Consider that value in R2 is #$04

```
WITH R2         ;Source and Destination are R2
LSR             ;Logical shift right two times
WITH R2         ;Source and Destination are R2 (again)
LSR             ;Now the value will be #$01
FROM R2         ;R2 is Source
STW (R0)        ;Store whatever is in R2 to R0
```

Unlike ASR, bit 15 will always be zero while bit 0 will be still transferred to the carry flag, thus not maintaining the MSB.

Consider that value in R1 is #$8000

```
WITH R1         ;Source and Destination are R1
ROR             ;Rotate bit two times
WITH R1         ;Source and Destination are R1 (again)
ROR             ;Now the value will be #$2000
FROM R1         ;R1 is Source
STW (R0)        ;Store whatever is in R1 to R0
```

With rotation operations, bit 15 and bit 0 will be always affected, since bit now rotates

rather than just being shifted around, so, anything in carry flag will be rotated to bit 15 while anything on bit 0 will be rotated to the carry flag.

Consider that value in R1 is #$C000

```
WITH R1         ;Source and Destination are R1
ROL             ;Rotate bit two times
WITH R1         ;Source and Destination are R1 (again)
ROL             ;Now the value will be #$03
FROM R1         ;R1 is Source
STW (R0)        ;Store whatever is in R1 to R0
```

The same story applies to ROL, except that carry flag will rotate to bit 0 and bit 15 will be rotated to carry flag.

The Multiplication operations are different than those bitwise operations, apart from extra steps that must be done for it to be properly completed, Multiplication doesn't use ALU for it's operations, instead, it uses a Multiplication Unit which runs at a different clock speed than the rest of GSU (which you can configure it by setting the register, just check CFGR register at Chapter 2), either you can make it run at the same speed clock of the GSU or halve it, in any case, I present you examples of those Multiplication operations:

Consider R5 as #$52CF and R1 as #$63CF

```
FROM R5         ;R5 is source
TO R2           ;R2 is destination
MULT R1         ;When MULT R1 is executed, it performs an 8-bit multiplication, so R5*R1
```
becomes #$0961 and store the result to R2.

Low bytes from both source and specified register are multiplied, using signed numbers, so we get this number.

Consider R5 as #$95C6

```
WITH R5         ;R5 is source and destination
MULT #9         ;When MULT #9 is executed, it performs an 8-bit multiplication, so R5*9
```
becomes #$FDF6 and store the result to R2.

It works the same way as MULT Rn except that you can specify values to multiply with the source register and store that into the destination register, remember to take care about the signedness of the MULT/LMULT/FMULT operations, UMULT doesn't use signedness so you can get “non-negative” values, beware!

Now, we're going to work with UMULT instruction, which is mostly MULT except that it does unsigned multiplication, the difference of signedness is that signed data holds both negative and positive numbers (E.g. $FF being -1) while unsigned holds positive only numbers (E.g. $FF being 255), here's an example:

Consider R5 as #$364F and R1 as #$B2CF

FROM R5 ;R5 is source
TO R2 ;R2 is destination
UMULT R1 ;When UMULT R1 is executed, it performs an 8-bit unsigned multiplication, so R5*R1 becomes #$3FE1 and store the result to R2.

As for 16-bit multiplication, manual states that 8-bit multiplication is done 4 times, and, the result is 32-bit, so you need to take extra care about the register used, as the default for this kind of multiplications use register R4, so you can't use this register as destination, otherwise, the result will be lost, remember this when using FMULT. Also, R6 is used as a source for multiplication along with the specified source register, so remember that too!

To understand how 16-bit multiplication works, we'll start with LMULT and then we'll go to FMULT, just remember that 16-bit calculations are done using signed data, that R6 is used as a source register for multiplication along with the specified source register and if messing with FMULT, you can't use R4 as a destination, here LMULT/FMULT examples:

Consider R9 as #$B556 and R6 as #$DAAB

FROM R9 ;Set source as R9
TO R8 ;Set destination as R8
LMULT ;Multiply R9 with R6 and store the upper 16 bits of result to the destination register while the lower 16 bits of result are stored in R4. If bit 15 of R6 is set, then carry flag is also set. The result will be that R8 will become #$9AE4 while R4 becomes #$5C72.

Warning: You can't use R4 with LMULT or the result will be invalid!

Consider R9 as #$4AAA and R6 as #$DAAB

FROM R9 ;Set source as R9
TO R8 ;Set destination as R8
FMULT ;Multiply R9 with R6 and store the upper 16 bit of result, if bit 15 of 32 bit result is set, so carry flag will be. R4 can't be assigned as a destination or the result will be lost.

As you can see, FMULT does store only the half (that's why it's called Fractioned Multiplication) to the destination register, which can't be R4 and store the bit 15 of the 32 bit result in carry flag. That's the major difference against the LMULT.

When using multiplication and you aren't sure what to do, check each step carefully and remember about signedness when using multiplication operations, that all 16 bit and 8 bit MULT multiplication does use signed numbers, UMULT doesn't, so take that in consideration!

Chapter 8: Bitwise Operations

GSU can do bitwise operations like SNES and it even has similar instructions for that, and unlike they say, bitwise operations aren't hard, however, they are complicated and requires logic.

We need to know that by bit, we mean **0 and 1**. **0 means FALSE while 1 means TRUE**.

GSU counts with 9 different opcodes for bitwise operations and we'll talk about each one, so don't worry, just keep reading every instruction will be explained, to start the NOT opcode.

NOT is a very simple instruction, it inverts every bit in the register like this one for example:

```
WITH R4     ;Set source and destination as R4 – R4 is #$5555
NOT         ;Invert every bit so 0101 0101 0101 0101 becomes 1010 1010 1010 1010 or #$AAAA
```

As you can see, NOT simply inverts every bit in the register, nothing more and nothing less.

Now, we'll start the AND opcode, AND is a very simple instruction too as you compare the values and if the bits match, the bit value is maintained while the other is discarded, here's some examples:

```
FROM R5     ;R5 is source – R5 is #$3015
TO R1       ;R1 is destination
AND R3      ;AND R3 with R3 being #$ABCD so do R5 AND R3
            ;R5 – 0011 0000 0001 0101; R3 – 1010 1011 1100 1101
            ;R1 – 0010 0000 0000 0101 - #$2005

WITH R10    ;R10 is source and destination – R10 is #$3212
AND #6      ;AND #6 so do R10 AND #6
            ;R10 – 0011 0010 0001 0010; #6 – 0000 0000 0000 0110
            ;R10 – 0000 0000 0000 0010 - #$2
```

The OR instruction works the opposite way as AND, if the bits doesn't match, then the value is maintained and isn't discarded, here's more examples:

```
FROM R7     ;R7 is source – R7 is #$3015
TO R1       ;R1 is destination
OR R3       ;OR R3 with R3 being #$ABCD so do R7 OR R3
            ;R7 – 0011 0000 0001 0101; R3 – 1010 1011 1100 1101
            ;R1 – 1011 1011 1101 1101 - #$BBDD

WITH R6     ;R6 is source and destination – R6 is #$3212
OR #6       ;OR #6 so do R6 OR #6
            ;R6 – 0011 0010 0001 0010; #6 – 0000 0000 0000 0110
```

```
                ;R6 – 0011 0010 0001 0110 - #$3216
```

Now we'll work with the XOR instruction, albeit similar of the OR instruction, this one takes in consideration that bits SHOULDN'T be matched, otherwise it's discarded, here's some examples:

```
FROM R7         ;R7 is source – R7 is #$3015
TO R1           ;R1 is destination
XOR R3          ;XOR R3 with R3 being #$ABCD so do R7 XOR R3
                ;R7 – 0011 0000 0001 0101; R3 – 1010 1011 1100 1101
                ;R1 – 1001 1011 1101 1000 - #$9BD8

WITH R6         ;R6 is source and destination – R6 is #$3212
XOR #6          ;XOR #6 so do R6 XOR #6
                ;R6 – 0011 0010 0001 0010; #6 – 0000 0000 0000 0110
                ;R6 – 0011 0010 0001 0100 - #$3214
```

And the last bitwise operations involves the use of BIC, BIC simply performs logical AND on corresponding bits of source register and the 1's complement of register specified in Rn, so this basically this is a AND+NOT, since bits are discarded if they match, here's some examples:

```
FROM R7         ;R7 is source – R7 is #$75CE
TO R1           ;R1 is destination
BIC R3          ;BIC R3 with R3 being #$3846 so do R7 BIC R3
                ;R7 – 0111 0101 1100 1110; R3 – 0011 1000 0100 0110
                ;R1 – 0100 0101 1000 1000 - #$4588

WITH R6         ;R6 is source and destination – R6 is #$364B
BIC #$F         ;BIC #$F so do R6 BIC #$F
                ;R6 – 0011 0110 0100 1011; #F – 0000 0000 0000 1111
                ;R6 – 0011 0110 0100 0000 - #$3640
```

Before we close this chapter, I should note a very simple instruction, called SWAP, it exchanges High byte with Low byte and vice-versa, here's an example:

```
WITH R6         ;R6 is source and destination – R6 is #$364B
SWAP            ;SWAP the bytes. Now R6 is #$4B36
```

With this last example, we finish this chapter with great enthusiasm since the chapter that everyone wants is getting closer and closer and it's the same for this tutorial, it's getting closer to the end.

Chapter 9: Pseudo-Indexing

SNES (and probably other kinds of ASMs) have an interesting feature called Indexing, it works like that (in SNES case), you've a value in either X or Y and then, you load the address in accumulator and if X or Y gets loaded by any value, you sum the value contained in those registers on accumulator. Since GSU doesn't have ANY Indexing instructions, it have to be done MANUALLY, unless you are some kind of a lazy person, you won't even notice this.

Here's a simple example of SNES indexing:

```
LDX $00             ;Load $00 to X
LDA Randomed,X      ;Load Randomed plus X
STA $FF             ;Store to $FF
RTS                 ;Return

Randomed: db $FF,$EE,$DD,$CC,$BB      ;A random table :3
```

If $00 loads a value of #$02, then load Randomed+$02, if loads a value of #$04, then load Randomed+$04 and so on.

As you can see, SNES uses several kinds of registers of indexing with special opcodes (the LDA $NN,x or LDA $NN,y for example) for that but GSU don't have none! How to do indexing then?

There's several ways of how indexing can be useful, like branches or loop codes but for now let us focus on the basic, to do a simple indexing loading from SRAM in GSU, we'll do like that:

```
IWT R0,#$8000       ;Load bank 8000 from SRAM (This contains our “address” to index)
IWT R3,#$8800       ;Load bank 8800 from SRAM (This contains our “address” to index too)

SNES:
ADD R1          ;Add R1+R0=R0 (let's suppose R1 is dynamic, this label will be used every time and address gets loaded also make $8000+$xx)
WITH R3         ;Set Source and Destination to R3
ADD R1          ;Add R1+R3=R3 (Make $8800+$xx)
TO R2           ;Set Destination to R2
LDW (R0)        ;Load R0 to R2 (this means that for example, if R1 was #$04, then it'll load $8000+$4 and so on)
WITH R2         ;Set Source and Destination to R2
STW (R3)        ;Store data from R2 to the address in R3 (same story as above, if R1 was #$04, then it'll store $8800+$4 and so on)
[...]
```

These codes can be a pain to work in GSU thanks to not it having special opcodes like SNES but it doesn't mean such thing can't be done, while the logic MAY seem complicated at

first, it's a matter of understanding how indexing works so you can replicate it on GSU, practice and then you can do indexing instructions just like you would do if you're working with SNES, just remember that it's all about logic and understanding what you need to do to load/store addresses just like the machine was supposed to do.

Chapter 10: Miscellaneous Opcodes

This chapter contains simple opcodes that doesn't deserve much explanation, some are useful and some aren't that useful but everything's here:

NOP: Unlike SNES version of NOP, this is simple used as a padding rather than making the CPU idle for nothing, this padding is because of pipeline and thus, NOP IS required when you use instructions that changes R15 or you don't want certain values to be changed but generally, this doesn't have to be used.

Macro "Instructions": There's a set of macro instructions that can automatically set things up for you, unlike normal instructions they simple changes how the assembler sees the code you inserted, unless you are lazy enough to do macro instructions do the work, then these instructions aren't useful at all, albeit they are explained in the Mnemonics section.

LINK #n: If you remember the subroutine section, you may remember seeing two types of return "input", one uses IWT while the other uses LINK. The main difference is that link is smaller, faster but it's direction can only go from 1 to 4, making this instruction quite useless.

LMS Rn,(yy): LMS does the same as LM, except that it uses short addressing and needs to use even addresses, while this can be useful, it's very limited so it's not a very useful instruction to meddle with.

SBK: If you have used LM or LMS once and you need to save data to the same address, you DON'T have to re-input the same address again, unless you have used another LM or LMS, use SBK to save data to the specified address, being faster and smaller! So this instruction is very useful albeit it's rarely used.

SMS (yy),Rn: The same story as LMS, this instruction works like SM, except it uses short addressing and needs the address to be even so this can be annoying and not that useful, specially if you count the use of SBK.

ALT1 – ALT2 – ALT3: This "changes" how CPU sees some opcodes since it enables the use of other forms of instructions such as ADD for ADC, MULT Rn to MULT #n to name a few, what makes this instruction quite useless is that you don't have to specify it if you put the instruction that uses them, as the assembler will automatically input that for you, you can use that as a mean to reduce cycles if you take pipeline into account (see the tips section for that).

MOVE Rn',Rn: This copies the value from Rn to Rn', this instruction is useful and as such can be used for simple logical inputs, however, you can do the same by inputting WITH with FROM or TO (MOVE Rn',Rn uses TO while MOVES Rn',Rn uses FROM and MOVE and MOVES use this scheme automatically just like the ALT instructions).

MOVES Rn',Rn: This copies the value from Rn to Rn' and set flags, this instruction is similar of MOVE and as noted, the only difference is that this one sets flags when needed, both instructions are useful albeit this one can be more useful if some flag operations are

needed.

CACHE: This instruction is amazing, this instruction, when executed, transfers data (temporarily) to a 512 byte Cache RAM which enables the use of a very high speed RAM, thus augmenting even more the power of processing, this instruction is simple enough with some details to be aware simply read the manual for more info.

DIV2: This instruction is USELESS. This is simply an ASR with a check to see if a number is -1. If the number is -1, then set it to 0, otherwise, ASR it. (Unless you want to do a check from it, you'll be wasting one byte and 3 extra cycles)

LOOP: There's no much to explain about LOOP opcode, except that you set R12 as the loop counter and R13 as the repeat address, every time LOOP is executed, R12 is decremented and R13 goes to R15.

SEX: (Don't laugh in this one) SEX is a very useless instruction, albeit it has it's uses, in general it doesn't do much, what it does it grabs the bit 7 of source and stores it from bit 8 to bit 15 of destination and loads the low byte of destination.

STOP: It's a very simple yet extremely important opcode, if GSU executes this instruction, G flag is cleared, IRQ is generated and the clock is stopped, giving SNES operation to work, it may not look it but it's USEFUL, otherwise GSU would be executing forever and SNES would stall.

Chapter 11: Bitmap Emulation and Plot Processing

In this chapter, we're going to work with the most prominent feature of GSU, which is the plot feature that made this chip memorable by it's released titles, such as Doom or Star Fox.

As noted above, GSU have powerful mathematical instructions, powerful clock speed, reduced instructions but even with that, it have a very special feature and it is the Bitmap Emulation and the Plot Processing.

What's the Bitmap Emulation and the Plot Processing?

Since SNES PPU can't perform graphical processing such as placing a point, drawing a line or painting a plane and these are bit-mapped graphics, PPU can't handle the conversion necessary for that without slowing down, so GSU is responsible to convert the bitmap data to character data, so SNES can upload that to VRAM thus effectively emulating bitmap.

So, what's the benefit of bitmap emulation?

You can perform graphical enhancement to your game, and this includes 2D or even 3D enhancement, such as rotation, resize or even the construction of polygon graphics!

And what we're going to learn in this chapter?

We're going to learn the very basics, from every instruction used to how to perform the bitmap emulation by plotting a simple trapezoid character. This simple tutorial will open doors for more advanced operations that YOU, the user, should try for yourself.

First of all, we need to know the instructions we are going to use for plot operation and they are: *COLOR, PLOT, CMODE, RPIX* and *GETC*. These instructions will be briefly explained and then, in this same chapter, we'll discuss each one in depth, just for summary sake.

The *COLOR* instruction loads the value in Source Register to the Color register, when you plot the data, it converts it to character data format, thus being readable by SNES. Usually, you change the color having the palette in mind, including color 0, unless you use dithering and/or transparency.

The *GETC* instruction does the same thing as the *COLOR* instruction, except that it doesn't need to load any register, instead, it uses ROM buffer to transfer data to color register.

The *CMODE* is a very special instruction, in fact, for the plot to even work, this instruction should be executed BEFORE any plot instruction or color instruction happens at all, this instruction sets everything needed for the plot to function, such as Transparency, Dither, OBJ Mode and so on, this will be explained further in this chapter.

The *RPIX* instruction is essentially the opposite of the *PLOT* instruction, by utilizing the R1

and R2 registers to load the desirable pixel and load the color of that specified pixel, *RPIX* may be used to flush unwanted data of the PLOT hardware.

And finally, the most important instruction of all, the *PLOT* instruction. GSU wouldn't be GSU if the PLOT instruction wasn't included in it's architecture, without further ado, PLOT is a simple instruction that plots the color data contained in the color register, grabs X/Y coordinates and plots the data to RAM, it does calculations and compares newer values for X/Y, also, it automatically increments the value for X, so you don't have to worry about drawing a line if you wish. In this chapter, we'll cover the basic operations needed for PLOT instruction and how to make a single trapezoid using PLOT.

Chapter 12: SNES Side Operations

This is the last obligatory chapter for you to follow if you want to learn how to do GSU works. Well, so far, you learned how to do every major operations on GSU and you can even interpret instances of GSU code and even do your own code but the major question you may ask, “How to make this thing work?”...

You know the basics, you can see how GSU ASM works but still you haven't make it work, what's wrong?

In this chapter, you'll learn how to do the “SNES Side Operations”, the last step of learning how to make GSU fully work on your game and/or hack.

To start, we must prepare the ROM to “receive” GSU, in it's normal form, the emulator or the SNES itself won't recognize the presence of the processor, all because we haven't modified several instances of ROM header so it'll still read the ROM as normal instead of a GSU included one, to modify, grab the hex editor of your choice or do an ASM file for you, every choice is a good choice as long you know what to do and as long you follow the directions right.

The header must be edited for 3 things:

1. Enable and fix the Expansion RAM Size
2. “Disable” SRAM
3. Activate the coprocessor itself

To do that, we must find and edit (all values uses headered addressing):

$00:FFBD to #$00 (None), #$01 (16KBit), #$03 (64KBit), #$05 (256Kbit), #$06 (512KBit), #$07 (1Mbit) – This controls the Expansion RAM Size

$00:FFDA to #$33 – If value don't get modified, ROM will get a fixed value of SRAM of 256Kbit.

From **$00:FFB6** to **$00:FFBC** must be #$00 – Unknown but probably messes with SRAM size too.

$00:FFD6 to #$13 (ROM + Coprocessor), #$14 (ROM + Coprocessor + RAM), #$15 (ROM + Coprocessor + RAM + Battery) – It is recommendable to set it at #$15 for better results (it activates GSU-2)

$00:FFD8 to #$00 (“Disable” SRAM) – According to manual, SRAM settings here is replaced by the Expansion RAM Size and therefore must be zeroed.

After you choose the options that suits your needs and you first open your ROM after the modifications, you'll see that GSU (Super FX) gets recognized but be warned, if emulator

recognizes more than 2MB of data and/or wrong mappings, GSU will be disabled and all operations related to it will fail (read: crash)

You see that GSU is recognized but it doesn't mean we can use it right away, if we use carelessly, it may even crash in real hardware! So we must do a few steps and after this, GSU can be used it without any worry, as long you remember it's limitations.

To start, after we got the ROM nice and ready for GSU, we gotta set the Dummy Vectors, in case you already forgot, go to Chapter 2 and read the addresses needed for you to change, in any case, you can set either that way or modify the entire Vector Info for your needs, like setting it in RAM just in case. You don't need but you can change from **$00:FFE0** to **$00:FFFF** (Vector Info: Native and Emulation modes respectively, usually only the Emulation is used only, apart from RESET)

By doing that, you'll assure that SNES can perform reads on interrupts on WRAM while GSU uses the ROM/RAM accordingly to it's needs. After setting the Vector Info, we must upload our IRQ/NMI to WRAM areas, so we can set a jump in those Dummy Vectors, simply that.

Now that we've set the Dummy Vectors, uploaded IRQ/NMI and set the Dummy Vectors to jump on desired places on WRAM, there's two more things to do, upload the Execution code and set the GSU to properly operate.

Considering you know how to upload the Execution Code to WRAM, all you need is to set the registers to operate and then jump to the routine, for that, consider this example:

Chapter 13: Tips, Hints, Cheats and Mnemonics

In this chapter you'll learn tips and hints that will help you in creating codes for GSU, this chapter is simple and entirely optional, but, if you want to get extra knowledge, you may want to read this one, some hints are obvious, others aren't.

Fact 1:

One SNES machine cycle for SlowROM, equals to approximately 0.3731 µs, one SA-1 machine cycle equals to approximately 0.0931 µs while one GSU machine cycle for 21.4 MHz equals to approximately 0.0467 µs.

Tip 1:

I'll do a mnemonics list for every opcode existent in GSU, detailing it not how it works but how it's read and how it's described, how it changes flags and how operator functions, since there isn't NO documented mnemonics list for GSU and in a specific way rather than functionally (as the manual do), I'll do that manually with every description needed, so don't worry, if you want how to read/interpret how the processor would interpret that too, enjoy:

Assembler Example	Hexadecimal Value	Bytes	Cycles
ADC Add with Carry [Flags affected: B[1], ALT1[1], ALT2[1],O/V[2], S[3], CY[4], Z[5]]			
ADC Rn	5n n being from ($0 to $F)	2[6]	ROM: 6 RAM: 6 C-RAM: 2
ADC #n	5n n being from ($0 to $F)	2[7]	ROM: 6 RAM: 6 C-RAM: 2
ADD Add [Flags affected: B[1], ALT1[1], ALT2[1],O/V[2], S[3], CY[4], Z[5]]			
ADD Rn	5n n being from ($0 to $F)	1	ROM: 3 RAM: 3 C-RAM: 1
ADD #n	5n n being from ($0 to $F)	2[8]	ROM: 6 RAM: 6 C-RAM: 2
ALT Alter Instruction (Flag Prefix) [Flags affected: ALT1, ALT2]			
ALT1	3D	1	ROM: 3 RAM: 3 C-RAM: 1
ALT2	3E	1	ROM: 3 RAM: 3 C-RAM: 1
ALT3	3F	1	ROM: 3 RAM: 3 C-RAM: 1
AND AND Register with Memory [Flags affected: B[1], ALT1[1], ALT2[1], S[3], Z[5]]			
AND Rn	7n n being from ($0 to $F)	1	ROM: 3 RAM: 3 C-RAM: 1
AND #n	7n n being from ($0 to $F)	2[8]	ROM: 6 RAM: 6 C-RAM: 2
ASR Arithmetic Register or Memory Shift Right [Flags affected: B[1], ALT1[1], ALT2[1], S[3], CY[9], Z[5]]			
ASR	96	1	ROM: 3 RAM: 3 C-RAM: 1
BCC Branch if Carry Clear [Flags affected: None]			
BCC Label	0C	2	ROM: 6 RAM: 6 C-RAM: 2
BCS Branch if Carry Set [Flags affected: None]			
BCS Label	0D	2	ROM: 6 RAM: 6

			C-RAM: 2
BEQ Branch if Equal [Flags affected: None]			
BEQ Label	09	2	ROM: 6 RAM: 6 C-RAM: 2
BGE Branch if Greater or Equal [Flags affected: None]			
BGE Label	07	2	ROM: 6 RAM: 6 C-RAM: 2
BIC Bit Clear Mask [Flags affected: B[1], ALT1[1], ALT2[1], S[3], Z[5]]			
BIC Rn	7n n being from ($0 to $F)	2[6]	ROM: 6 RAM: 6 C-RAM: 2
BIC #n	7n n being from ($0 to $F)	2[7]	ROM: 6 RAM: 6 C-RAM: 2
BLT Branch if Less Than [Flags affected: None]			
BLT Label	06	2	ROM: 6 RAM: 6 C-RAM: 2
BMI Branch if Minus [Flags affected: None]			
BMI Label	0B	2	ROM: 6 RAM: 6 C-RAM: 2
BNE Branch if Not Equal [Flags affected: None]			
BNE Label	08	2	ROM: 6 RAM: 6 C-RAM: 2
BPL Branch if Plus [Flags affected: None]			
BPL Label	0A	2	ROM: 6 RAM: 6 C-RAM: 2
BRA Branch Always [Flags affected: None]			
BRA Label	05	2	ROM: 6 RAM: 6 C-RAM: 2
BVC Branch if Overflow Clear [Flags affected: None]			
BVC Label	0E	2	ROM: 6 RAM: 6 C-RAM: 2
BVS Branch if Overflow Set [Flags affected: None]			
BVS Label	0F	2	ROM: 6 RAM: 6 C-RAM: 2

CACHE Set and Upload to Cache [Flags affected: B[1], ALT1[1], ALT2[1]]			
CACHE	02	1	ROM: 3~4 RAM: 3~4 C-RAM: 1
CMODE Set Color Mode to Plot Options Register [Flags affected: B[1], ALT1[1], ALT2[1]]			
CMODE	4E	2^6	ROM: 6 RAM: 6 C-RAM: 2
CMP Compare Register with Memory [Flags affected: B[1], ALT1[1], ALT2[1],O/V[2], S[3], CY[4], Z[5]]			
CMP Rn	6n n being from ($0 to $F)	2^7	ROM: 6 RAM: 6 C-RAM: 2
COLOR Load From Register to Color Register [Flags affected: B[1], ALT1[1], ALT2[1]]			
COLOR	4E	1	ROM: 3 RAM: 3 C-RAM: 1
DEC Decrement [Flags affected: B[1], ALT1[1], ALT2[1], S[3], Z[5]]			
DEC Rn	En n being from ($0 to $E)	1	ROM: 3 RAM: 3 C-RAM: 1
DIV2 Divide by Two [Flags affected: B[1], ALT1[1], ALT2[1], S[3], CY[9], Z[5]]			
DIV2	96	2^6	ROM: 6 RAM: 6 C-RAM: 2
FMULT Fractional Multiplication [Flags affected: B[1], ALT1[1], ALT2[1], S[3], CY[10], Z[11]]			
FMULT	9F	1	ROM: 11 or 7 RAM: 11 or 7 C-RAM: 8 or 4
FROM From Register (Set Destination Register) (Register Prefix) [Flags affected: None]			
FROM Rn	Bn n being from ($0 to $F)	1	ROM: 3 RAM: 3 C-RAM: 1
GETB Get Byte Data from ROM Buffer [Flags affected: B[1], ALT1[1], ALT2[1]]			
GETB	EF	1	ROM: 3~8 RAM: 3~9 C-RAM: 1~6
GETBH Get High Byte Data from ROM Buffer [Flags affected: B[1], ALT1[1], ALT2[1]]			
GETBH	EF	2^6	ROM: 6~10 RAM: 6~9 C-RAM: 2~6
GETBL Get Low Byte Data from ROM Buffer [Flags affected: B[1], ALT1[1], ALT2[1]]			
GETBL	EF	2^8	ROM: 6~10 RAM: 6~9 C-RAM: 2~6

GETBS Get Signed Byte Data from ROM Buffer [Flags affected: B^{1}, $ALT1^{1}$, $ALT2^{1}$]			
GETBS	EF	2^{7}	ROM: 6~10 RAM: 6~9 C-RAM: 2~6
GETC Get Byte Data from ROM Buffer to Color Register [Flags affected: B^{1}, $ALT1^{1}$, $ALT2^{1}$]			
GETC	DF	1	ROM: 3~10 RAM: 3~9 C-RAM: 1~6
HIB Get High Byte [Flags affected: B^{1}, $ALT1^{1}$, $ALT2^{1}$, S^{12}, Z^{13}]			
HIB	C0	1	ROM: 3 RAM: 3 C-RAM: 1
IBT Immediate Byte Transfer to Register [Flags affected: B^{1}, $ALT1^{1}$, $ALT2^{1}$]			
IBT Rn,#$pp	An n being from ($0 to $F) pp being from ($00 to $FF)	2	ROM: 6 RAM: 6 C-RAM: 2
INC Increment [Flags affected: B^{1}, $ALT1^{1}$, $ALT2^{1}$, S^{3}, Z^{5}]			
INC Rn	Dn n being from ($0 to $F)	1	ROM: 3 RAM: 3 C-RAM: 1
IWT Immediate Word Transfer to Register [Flags affected: B^{1}, $ALT1^{1}$, $ALT2^{1}$]			
IWT Rn,#$xx	Fn n being from ($0 to $F) xx being from ($0000 to $FFFF)	3	ROM: 9 RAM: 9 C-RAM: 3
JMP Jump [Flags affected: B^{1}, $ALT1^{1}$, $ALT2^{1}$]			
JMP Rn	9n n being from ($8 to $D)	1	ROM: 3 RAM: 3 C-RAM: 1
LDB Load Byte from SRAM [Flags affected: B^{1}, $ALT1^{1}$, $ALT2^{1}$]			
LDB (Rm)	4m m being from ($0 to $B)	2^{6}	ROM: 11 RAM: 13 C-RAM: 6
LDW Load Word from SRAM [Flags affected: B^{1}, $ALT1^{1}$, $ALT2^{1}$]			
LDW (Rm)	4m m being from ($0 to $B)	1	ROM: 10 RAM: 12 C-RAM: 7
LINK Link Return Address to R11 [Flags affected: B^{1}, $ALT1^{1}$, $ALT2^{1}$]			
LINK	9n n being from ($1 to $4)	1	ROM: 3 RAM: 3 C-RAM: 1

LJMP Long Jump [Flags affected: B[1], ALT1[1], ALT2[1]]			
LJMP Rn	9n n being from ($8 to $D)	2[6]	ROM: 6 RAM: 6 C-RAM: 2
LM Load from SRAM to Register [Flags affected: B[1], ALT1[1], ALT2[1]]			
LM Rn,(xx)	Fn n being from ($0 to $F) xx being from ($0000 to $FFFF)	4[6]	ROM: 20 RAM: 21 C-RAM: 11
LMS Load from SRAM to Register using Short Address [Flags affected: B[1], ALT1[1], ALT2[1]]			
LMS Rn,(yy)	An n being from ($0 to $F) yy being from ($00 to $FF)	3[6]	ROM: 17 RAM: 17 C-RAM: 10
LMULT Long Multiplication [Flags affected: B[1], ALT1[1], ALT2[1], S[3], CY[14], Z[15]]			
LMULT	9F	2[6]	ROM: 10 or 14 RAM: 10 or 14 C-RAM: 5 or 9
LOB Get Low Byte [Flags affected: B[1], ALT1[1], ALT2[1], S[16], Z[17]]			
LOB	9E	1	ROM: 3 RAM: 3 C-RAM: 1
LOOP Loop Routine [Flags affected: B[1], ALT1[1], ALT2[1], S[18], Z[19]]			
LOOP	3C	1	ROM: 3 RAM: 3 C-RAM: 1
LSR Logical Shift Memory or Register Right [Flags affected: B[1], ALT1[1], ALT2[1], S[1], CY[9], Z[5]]			
LSR	03	1	ROM: 3 RAM: 3 C-RAM: 1
MERGE Merge High Byte of R7 to High Byte and High Byte of R8 to Low Byte for Destination Register [Flags affected: B[1], ALT1[1], ALT2[1],O/V[20], S[21], CY[22], Z[23]]			
MERGE	70	1	ROM: 3 RAM: 3 C-RAM: 1
MOVE Move from Register n' to Register n [Flags affected: B[1], ALT1[1], ALT2[1]]			
MOVE Rn, Rn'	2n' n' being from ($0 to $F) 1n n being from ($0 to $F)	2[29]	ROM: 6 RAM: 6 C-RAM: 2
MOVES Move from Register n' to Register n and Set Flags [Flags affected: B[1], ALT1[1],			

ALT2[1],O/V[24], S[25], Z[26]]			
MOVES Rn, Rn'	2n' n' being from ($0 to $F) Bn n being from ($0 to $F)	2^{29}	ROM: 6 RAM: 6 C-RAM: 2
MULT Multiply 8-bit Memory [Flags affected: B[1], ALT1[1], ALT2[1], S[3], Z[5]]			
MULT Rn	8n n being from ($0 to $F)	1	ROM: 3 or 5 RAM: 3 or 5 C-RAM: 1 or 2
MULT #n	8n n being from ($0 to $F)	2^{8}	ROM: 6 or 8 RAM: 6 or 8 C-RAM: 2 or 3
NOP No Operation [Flags affected: B[1], ALT1[1], ALT2[1]]			
NOP	01	1	ROM: 3 RAM: 3 C-RAM: 1
NOT Perform Logic Negation a.k.a Invert Bits [Flags affected: B[1], ALT1[1], ALT2[1], S[3], Z[5]]			
NOT	4F	1	ROM: 3 RAM: 3 C-RAM: 1
OR Or Register with Memory [Flags affected: B[1], ALT1[1], ALT2[1], S[3], Z[5]]			
OR Rn	Cn n being from ($1 to $F)	1	ROM: 3 RAM: 3 C-RAM: 1
OR #n	Cn n being from ($1 to $F)	2^{8}	ROM: 6 RAM: 6 C-RAM: 2
PLOT Plot color to coordinates X and Y [Flags affected: B[1], ALT1[1], ALT2[1]]			
PLOT	4C	1	ROM: 3~48 RAM: 3~51 C-RAM: 1~48
RAMB Set RAM Bank [Flags affected: B[1], ALT1[1], ALT2[1]]			
RAMB	DF	2^{8}	ROM: 6 RAM: 6 C-RAM: 2
ROL Rotate Left [Flags affected: B[1], ALT1[1], ALT2[1], S[3], CY[27], Z[5]]			
ROL	04	1	ROM: 3 RAM: 3 C-RAM: 1
ROMB Set ROM Bank [Flags affected: B[1], ALT1[1], ALT2[1]]			
ROMB	DF	2^{7}	ROM: 6 RAM: 6 C-RAM: 2

ROR Rotate Right [Flags affected: B[1], ALT1[1], ALT2[1], S[3], CY[27], Z[5]]			
ROR	97	1	ROM: 3 RAM: 3 C-RAM: 1
RPIX Read Pixel Color [Flags affected: B[1], ALT1[1], ALT2[1], S[3], Z[5]]			
RPIX	4C	2[6]	ROM: 24~80 RAM: 24~78 C-RAM: 20~74
SBC Subtract with Carry [Flags affected: B[1], ALT1[1], ALT2[1],O/V[2], S[3], CY[28], Z[5]]			
SBC Rn	6n n being from ($0 to $F)	2[6]	ROM: 6 RAM: 6 C-RAM: 2
SBK Save Bulk Data to SRAM (Last Address Used) [Flags affected: B[1], ALT1[1], ALT2[1]]			
SBK	90	1	ROM: 3~8 RAM: 7~11 C-RAM: 1~6
SEX Signed Expansion [Flags affected: B[1], ALT1[1], ALT2[1], S[3], Z[5]]			
SEX	95	1	ROM: 3 RAM: 3 C-RAM: 1
SM Save to SRAM [Flags affected: B[1], ALT1[1], ALT2[1]]			
SM (xx), Rn	Fn n being from ($0 to $F) xx being from ($0000 to $FFFF)	4[8]	ROM: 12~17 RAM: 16~20 C-RAM: 4~9
SMS Save to SRAM Using Short Address [Flags affected: B[1], ALT1[1], ALT2[1]]			
SMS (yy), Rn	An n being from ($0 to $F) yy being from ($00 to $FF)	3[8]	ROM: 9~14 RAM: 13~17 C-RAM: 3~8
STB Store byte to SRAM [Flags affected: B[1], ALT1[1], ALT2[1]]			
STB (Rm)	3m m being from ($0 to $B)	2[6]	ROM: 6~9 RAM: 8~14 C-RAM: 2~5
STOP Stop the Clock [Flags affected: B[1], ALT1[1], ALT2[1],G[1]]			
STOP	00	1	ROM: 3 RAM: 3 C-RAM: 1
STW Store Word to SRAM [Flags affected: B[1], ALT1[1], ALT2[1]]			
STW (Rm)	3m m being from ($0 to $B)	1	ROM: 3~8 RAM: 7~11 C-RAM: 1~6

SUB Subtract [Flags affected: B[1], ALT1[1], ALT2[1], O/V[2], S[3], CY[28], Z[5]]			
SUB Rn	6n n being from ($0 to $F)	1	ROM: 3 RAM: 3 C-RAM: 1
SUB #n	6n n being from ($0 to $F)	2^8	ROM: 6 RAM: 6 C-RAM: 2
SWAP Swap Byte Positions [Flags affected: B[1], ALT1[1], ALT2[1], S[3], Z[5]]			
SWAP	4D	1	ROM: 3 RAM: 3 C-RAM: 1
TO To Register (Register Prefix) (Set Destination Register) [Flags affected: None]			
TO Rn	1n n being from ($0 to $F)	1	ROM: 3 RAM: 3 C-RAM: 1
UMULT Unsigned 8-bit Multiplication [Flags affected: B[1], ALT1[1], ALT2[1], S[3], Z[5]]			
UMULT Rn	8n n being from ($0 to $F)	2^6	ROM: 6 or 8 RAM: 6 or 8 C-RAM: 2 or 3
UMULT #n	8n n being from ($0 to $F)	2^7	ROM: 6 or 8 RAM: 6 or 8 C-RAM: 2 or 3
WITH Register as Source and Destination [Flags affected: B[30]]			
WITH Rn	2n	1	ROM: 3 RAM: 3 C-RAM: 1
XOR Exclusive OR [Flags affected: B[1], ALT1[1], ALT2[1], S[3], Z[5]]			
XOR Rn	Cn n being from ($0 to $F)	2^6	ROM: 6 RAM: 6 C-RAM: 2
XOR #n	Cn n being from ($0 to $F)	2^7	ROM: 6 RAM: 6 C-RAM: 2

1. Reset flags.
2. Set on signed overflow, else reset.
3. Set if result is negative, else reset.
4. Set on unsigned carry, else reset.
5. Set if result is zero, else reset.
6. Assemble ALT1 automatically.
7. Assemble ALT3 automatically.
8. Assemble ALT2 automatically.
9. Set if bit 0 in the source register is “1”, else reset.
10. Set when bit 15 of the result is “1”, else reset.
11. Set if upper 16 bit of result are zero, else reset.
12. Set if a negative number is loaded to the low byte of the destination register, else reset.
13. Set if zero is loaded to low byte of the destination register, else reset.

14. Set if Bit 15 of R6 is “1”, else reset.
15. Set if the destination register result is zero, else reset.
16. Set if low byte of source register is negative, else reset.
17. Set if low byte of the source register is zero, else reset.
18. Set if the register R12 is negative, else reset.
19. Set if the register R12 is zero, else reset.
20. Set if result of (B6 or B7 or B14 or B15) is “1”, else reset.
21. Set if the result of (B7 or B15) is “1”, else reset.
22. Set if the result of (B5 or B6 or B7 or B13 or B14 or B15) is “1”, else reset.
23. Set if the result of (B4 or B5 or B6 or B7 or B12 or B13 or B14 or B15) is “1”, else reset.
24. Set if bit 7 is “1”, else reset.
25. Set if bit 15 is “1”, else reset.
26. Set when data is zero, else reset.
27. Set if bit 15 in source register is “1”, else reset.
28. Set on unsigned overflow, else reset.
29. Assemble with WITH
30. Set B flag

Macro instructions

GSU counts with several macro instructions to help with the code, I'll describe them below.

LEA Rn, yyxx: This instruction is a simply alias for IWT, except it doesn't have the “#” signal.

MOVE Rn, #yyxx. This instruction is a conditional instruction that changes accordingly to what value you put here, in case it's a value ranging from $80 to $7F (or -128 to 127), then this instruction will be an IBT, otherwise, it would be an IWT.

MOVE Rn, (yyxx). This other instruction is also a conditional instruction that changes accordingly to what value you put here, in case it's a value ranging from $0000 to $01FF and it's even, use the LMS instruction, otherwise, use LM.

MOVE (yyxx), Rn. Accordingly to what value you put here, if it's a value ranging from $0000 to $01FF and it's even, use the SMS instruction, otherwise, use SM.

MOVEB Rn, (Rn'). This instruction is a different conditional one, instead of values, it relates to registers, it works for loading byte values, if Rn is R0, then only LDB (Rn') is used, otherwise uses TO Rn LDB (Rn').

MOVEB (Rn'), Rn. Same as the instruction above but relates to saving, if Rn is R0, then only STB (Rn') is used, otherwise uses FROM Rn STB (Rn').

MOVEW Rn, (Rn'). Same as the instruction above but relates to loading word values, if Rn is R0, then only LDW (Rn') is used, otherwise uses TO Rn LDW (Rn').

MOVEW (Rn'), Rn. Same as the instruction above but relates to saving word values, if Rn is R0, then only STW (Rn') is used, otherwise uses FROM Rn STW (Rn').

That's all the macro opcodes existent in GSU assembly, it can be handy for those who need a small and clean code rather than having to work with direct opcodes.

Chapter 14: Dictionary, FAQ and End Notes

In this last chapter, you'll know the meaning of commonly used words in this tutorial and in GSU manual, while I'll present commonly asked questions about GSU, assembly, hardware specifications and etcetera and the tutorial will be closed to the end notes.

A

Argonaut Games – Was a video game developer. Founded as Argonaut Software by Jez San (**Jeremy 'Jez' San)** in 1982 and in October 2004 was made defunct.

B

BRAM – Backup RAM. It's an unused RAM that would probably be related to storage of data in case of power loss. It's mapped to **$78:0000** and currently there are no cartridges with such settings.

C

Cycle – Unit to measure a time period that a processor executes a machine instruction.

D

DSP-1 – Digital Signal Processor – This processor allowed for fast vector-based calculations, bitmap conversions, both 2D and 3D coordinate transformations, and other functions except less powerful than GSU and didn't count with a PLOT instruction.

E

F

G

Game Pak ROM – It's the ROM of the cartridge, the manual states that this is the “ROM” area.

Game Pak RAM – It's the SRAM of cartridge actually it is the WRAM of GSU and “SRAM” for SNES. An unused type of RAM called, BRAM, was yet to be included but was scrapped.

GSU – Graphic Support processing Unit. It's the alternate name for Super FX.

H

I

J

K

L

M

Most Significant Bit – is the bit position in a binary number having the greatest value. MSB can also correspond to the sign bit of a signed binary number in one's or two's complement notation, "1" meaning negative and "0" meaning positive.

N

O

Opcode – Operation Code. It's the portion of a machine language instruction that specifies the operation to be performed.

P

Pipeline – The GSU pipeline system albeit does make instructions execute efficiently, doesn't execute it faster as stated in manual, although it increases the throughput for instruction execution, making it perform multiple operations in parallel.

Q

R

RISC – Reduced instruction set computing. It's a different CPU design that states that instructions are processed faster and lead to high performance based that an instruction should be simplified.

S

SNES – Super Nintendo Entertainment System. It's the short name for the console that was present in the 16 bit era of gaming.

T

U

V

W

X

Y

Z

References and Credits

GSU and other SNES related documents. Available at:

http://nocash.emubase.de/fullsnes.htm#snescartgsunprogrammablerisccpuakasuperfxmariochip10games

Accessed in 01/25/2014

GSU Patent:

San E. Jeremy *et al*, “External memory system having programmable graphics processor for use in a video game system of the like,” U.S. Patent 7 229 355, June 12, 2007.