What is ASM?

A program (games are programs, of course) can be roughly divided in two parts: code and resources. In a NDS ROM, **code files are the arm9.bin, arm7.bin and the overlay files**, while other files are resources.

However, the program code (C in this case) does not show as Game Freak wrote it but compiled. Compilation results in a degeneracy of the code into processor instructions, where all the function names are lost, or variables and structs don't exist anymore and are converted in registers and pointers. That is what we call **ASM**.

However, most of the code can be identified easily by the other pieces of code they call. In this regard, these fragments of code are called **functions** in a source code and **subroutines** in ASM, but they are equivalent: they can receive data (arguments), process it and return an output. Of course, like functions, there are subroutines that may not have neither input nor output. Except when the function uses the *inline* keyword, every function call in a source will be translated as a BL, BX or BLX in ASM. That's the most important thing for identifying code's purpose or writing new ASM code, as long as we know what does the called subroutine.

ASM

```
RAM:02043F68 ; Subroutine
RAM:02043F68
RAM:02043F68 sub_2043F68
RAM:02043F68 PUSH    {R3-R5,LR}
RAM:02043F6A MOVS    R4, R0
RAM:02043F6C MOVS    R1, R4
RAM:02043F6E ADDS    R1, #0x80
RAM:02043F70 LDR     R1, [R1]
RAM:02043F72 LDR     R5, [R1,#0xC]
RAM:02043F74 BL      sub_203FE24    ; sub-function1
RAM:02043F78 ADDS    R4, #0x80
RAM:02043F7A MOVS    R1, R0
RAM:02043F7C LDR     R0, [R4]
RAM:02043F7E BL      sub_204036C    ; sub-function2
RAM:02043F82 MOVS    R4, R0
RAM:02043F84 MOVS    R0, R5
RAM:02043F86 BL      sub_2027500    ; function2
RAM:02043F8A CMP     R0, #0
RAM:02043F8C BEQ     loc_2043F94
RAM:02043F8E MOVS    R0, #0
RAM:02043F90 STRH    R0, [R4]
RAM:02043F92 B       loc_2043FB8
RAM:02043F94 ; ---------------------------------------
RAM:02043F94
RAM:02043F94 loc_2043F94
RAM:02043F94 MOVS    R0, R5
RAM:02043F96 BL      sub_20274E0    ; function3
RAM:02043F9A CMP     R0, #0
RAM:02043F9C BNE     loc_2043FA4
RAM:02043F9E MOVS    R0, #1
RAM:02043FA0 STRH    R0, [R4]
RAM:02043FA2 B       loc_2043FB8
RAM:02043FA4 ; ---------------------------------------
RAM:02043FA4
RAM:02043FA4 loc_2043FA4
RAM:02043FA4 MOVS    R0, R5
RAM:02043FA6 BL      sub_2027520    ; function4
RAM:02043FAA CMP     R0, #0
RAM:02043FAC BEQ     loc_2043FB4
RAM:02043FAE MOVS    R0, #2
RAM:02043FB0 STRH    R0, [R4]
RAM:02043FB2 B       loc_2043FB8
RAM:02043FB4 ; ---------------------------------------
RAM:02043FB4
RAM:02043FB4 loc_2043FB4
RAM:02043FB4 MOVS    R0, #3
RAM:02043FB6 STRH    R0, [R4]
RAM:02043FB8
RAM:02043FB8 loc_2043FB8
RAM:02043FB8
RAM:02043FB8 MOVS    R0, #0
RAM:02043FBA POP     {R3-R5,PC}
RAM:02043FBA ; End of function sub_2043F68
```

C

```
static void Subroutine(int arg)
{
    int* variable1 = arg->fld1->fld2;
    uint16* variable2 = function1(arg);

    if (function2(variable1))
    {
        *variable2 = 0;
    }
    else if (function3(variable1) == FALSE)
    {
        *variable2 = 1;
    }
    else if (function4(variable1))
    {
        *variable2 = 2;
    }
    else
    {
        *variable2 = 3;
    }

    return 0;
}

inline void function1(int integer)
{
    return subfunction2(integer->fld1, subfunction1(integer));
}
```

(Note that, in the image above, *function1*() is an inline function and it is not branched to by the code, but directly compiled inside the invoking function)

The main difficulty here is to find the subroutines original names, as they are lost at compilation. The only way to have them would be having the original source code, or supposing them by analogy with a similar source code.

What are the code files?

The original source code ends compiled in 16-bit instructions (for these parts of the program that are compiled in THUMB mode) and 32-bit instructions (for ARM mode). This means 2 bytes and 4 bytes, respectively, for each instruction to be processed. Each different instruction will have a different 16-bit or 32-bit value (for example, **NOP** instruction is always C0 46 in THUMB mode, little endian). The code files in the ROM (arm9.bin and overlays are the most important ones) are in fact a bunch of 16-bit and 32-bit instructions, forming large byte sequences for each function/subroutine.

These byte sequences that form subroutines can be modified, but never expanded beyond its original size. That is because every subroutine coded in the code files interacts with other subroutines with both relative jumps/branches and absolute addresses. Editing every subroutine in the code files for fixing every branch is unapproachable, so the only way to write new subroutines is to find free space in the original code files, or finding a method for loading our data in a free RAM address.

In the code files we can also find predefined arrays that subroutines can use. These predefined arrays are also known as **tables** in the ROM hacking scene (for example, the type effectiveness table, the overworld table or the map headers).

The RAM memory

When the program (the game) starts, the very first thing that happens is the dump of the arm9.bin contents (in any case **decompressed**, even if the file is normally compressed in the ROM) in address 0x02000000 of the RAM memory. The processor can’t read nor store data in other place than the RAM memory, that’s why the code file contents need to be dumped in the RAM, so the processor can read and execute their instructions. It’s also the reason of the code instructions to refer other subroutines of the program by its position in the RAM memory and not by its position in the code files.

However, every compiled instruction of the program would take a lot of RAM memory if they were loaded at the same time. That’s why **overlayed** code files exist. The overlays are different files with different purposes in the program, but they are not always loaded in the RAM, but only when they are needed. In our case, the most significant cases are the overworld overlay set (some overlays that manage code that is only expected to run in the overworld) and the battle overlay set (overlays only expected to appear when the game is in a battle). That avoids a huge waste of memory, so more resources (images, data, 3D models) can be stored in the RAM.

Registers

The processor registers are the “variables” that can be operated in ASM, due to the inexistence of real variables in assembly. In both GBA and NDS they always have a **32-bit size** (4 bytes). In Thumb mode (the instruction set that we are going to use in 99.999% of the cases) we only have 8 common registers (from R0 to R7) and 3 special registers. Instructions always refer to a specific operation with determined register/s. The common registers are used for common arithmetic/logical operations or RAM memory access. However, special registers have specific roles and should not be used for other purposes. These are:

SP – **Stack pointer**: This register stores the current pointer of the stack in the RAM memory. We will cover the stack later, but we can define it as the memory region where unused register values are stored until they are loaded into a register and then operated. It is very important, because we couldn't do everything we want with only 8 registers for the whole program.

LR – Link register: This register is important when calling a subroutine (using BL or BLX), because it stores the pointer to the location where the program was before accessing that subroutine. In other words, it stores the pointer to where the program must return after calling a function. At the end of a subroutine, this register's value somehow must end in PC register.

PC – Program counter: This register stores the RAM address of the next instruction that will be executed by the processor.

Instructions

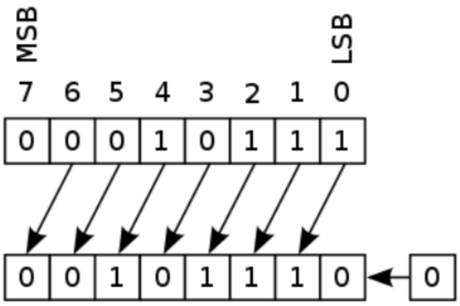
The most important instructions of THUMB mode are the following ones.

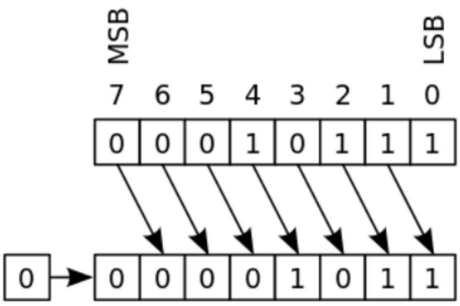
INSTR	PURPOSE	USAGE
NOP	Nothing (commonly used as a placeholder)	NOP
LSL	Logical shift of R_B, *value*/R_C bits to the left. Store result in R_A This is equivalent of multiplying R_B by 2^{value} or 2^{RC}	LSL R_A, R_B, *value* LSL R_A, R_B, R_C
LSR	Logical shift of R_B, *value*/R_C bits to the right. Store result in R_A This is equivalent of dividing R_B by 2^{value} or 2^{RC}	LSR R_A, R_B, *value* LSR R_A, R_B, R_C
ADD	Sum R_B and R_C, store result in R_A Sum *value* to R_A	ADD R_A, R_B, R_C ADD R_A, *value*
SUB	Subtract R_C to R_B, store result in R_A Subtract *value* to R_A	ADD R_A, R_B, R_C ADD R_A, *value*
MUL	Multiply R_B to R_A, store result in R_A	MUL R_A, R_B
AND	Makes a bitwise and between R_B and R_C, store result in R_A	AND R_A, R_B, R_C
ORR	Makes a bitwise or between R_B and R_C, store result in R_A	ORR R_A, R_B, R_C
EOR	Makes a bitwise xor between R_B and R_C, store result in R_A	EOR R_A, R_B, R_C
MOV	Loads *value* in R_A Loads R_B in R_A	MOV R_A, *value* MOV R_A, R_B
LDR	Loads 4 bytes from memory address [R_B + *value*] in R_A	LDR R_A, [R_B, *value*]
LDRH	Loads 2 bytes from memory address [R_B + *value*] in R_A	LDRH R_A, [R_B, *value*]
LDRB	Loads 1 byte from memory address [R_B + *value*] in R_A	LDRB R_A, [R_B, *value*]
STR	Stores 4 bytes of R_A in memory address [R_B + *value*]	STR R_A, [R_B, *value*]
STRH	Stores 2 bytes of R_A in memory address [R_B + *value*]	STRH R_A, [R_B, *value*]
STRB	Stores 1 byte of R_A in memory address [R_B + *value*]	STRB R_A, [R_B, *value*]
CMP	Compare R_A and R_B, updates internal CPU flags	CMP R_A, R_B
B	Execution jumps to *address*	B *address*
BEQ	Execution jumps to *address* if $R_A = R_B$ after a CMP	BEQ *address*
BNE	Execution jumps to *address* if $R_A \neq R_B$ after a CMP	BNE *address*
BCS	Execution jumps to *address* if $R_A >= R_B$ after a CMP	BCS *address*
BCC	Execution jumps to *address* if $R_A < R_B$ after a CMP	BCC *address*
BL	Saves the current PC value in LR, then jumps to *address*	BL address
BX	Execution jumps to R_A	BX R_A
BLX	Same as BL, but changes the instruction set (from ARM to THUMB mode or from THUMB to ARM mode)	BLX *address* BLX R_A
PUSH	Saves specified registers in the stack	PUSH {R_A-R_B}
PUSH	Loads specified registers from the stack	POP {R_A-R_B}

Logical and arithmetical instructions

These instructions operate with the current values in registers. Each instruction has a specific purpose and may come from different C source operations. As the table shows, the first register (R_A) is the only one that gets updated, while the others remain with the same value.

It is important to explain how the LSR and LSL instructions work. Both are logical shifts, so the bits of a register move to right or left as many places as specified in the instruction. This means that, for each position that the bits are moved to, the register's value gets multiplied or divided by 2.





The bits that overflow the 32-bit register size at the left (in a LSL) or at the right (in a LSR) are lost, so the information that these bits had cannot be recovered and are always filled with zeros.

Loading and writing instructions

These instructions can access to the RAM memory, load information from there and store it in registers so the processor can operate them. They can also write registers' values to the RAM memory (after processing them, for example).

MOV is the assignation instruction, as it can directly load a value from 0 to 255 in a register. It can also copy a register's value to another register. However, MOV cannot be used for loading a value higher than 256 in a register.

LDR and STR instructions can work with words (32-bit values), halfwords (16-bit values) or bytes (8-bit values), these last two with the LDRH/STRH and LDRB/STRB instructions. They need a RAM address, stored in the register they use, to work. Along with that register, they can also have a specified value that increase that address.

LDR has also a special purpose in most of subroutines. Remember that MOV could not load values higher than 255 in a register? LDR allows using PC as register, so it can load 32-bit values (4 bytes) near the current subroutine (the distance between the LDR instruction and the 4 loaded bytes are defined by the "increasing" value we mentioned). These data bytes are usually stored at the end of a subroutine and before the next subroutine starts. In this case, the LDR instruction is common to be represented as LDR R_A, =*value*.

Branching instructions

They allow the code to branch to different parts of the subroutine, or even call other subroutines.

There are different conditions for B instruction (as they can be seen in the table), all of them perform short jumps and usually inside the same subroutine. They need a CMP instruction for checking the condition between two register values. These instructions come from If, For and While structures in the C source.

We also have BL and BLX instruction, probably the most important ones. They perform big jumps (from the arm9 region to an overlay region, for example). They are also an exception of the THUMB mode, as they are the only instructions that are 32-bit long (unlike the other ones that are 16-bit long). As we explained in the introduction of this document, every BL and BLX instruction comes from a function call in the C source. In fact, we must keep seeing it like subroutine call instructions.

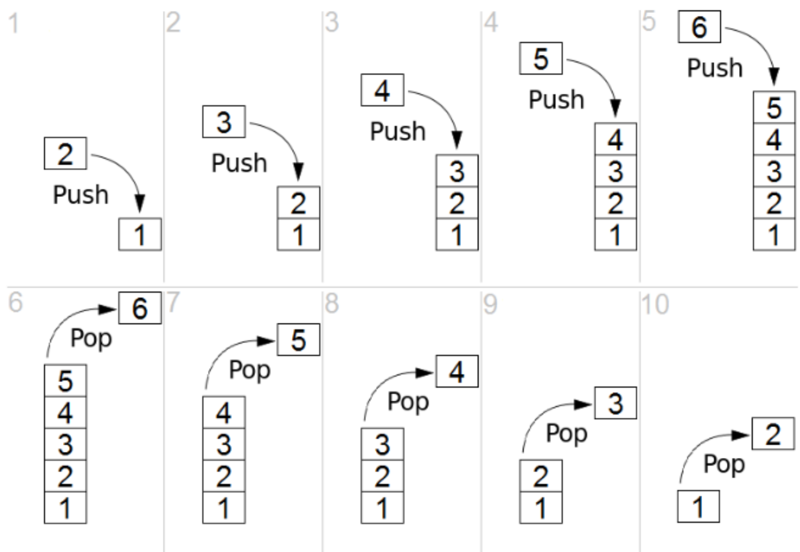
BLX also allows to switch between ARM and THUMB modes. It always changes the processor to THUMB mode if it was in ARM mode, or to ARM mode if it was in THUMB mode. Usually, everything in the program is compiled/encoded in THUMB mode except library subroutines (fixed- and floating-point functions, divmod functions, vector operation functions…) because they need a more powerful instruction set, so we will commonly see a BLX when a library function is called in the program.

BX (and BLX when uses a register instead of an offset) can also perform a instruction set switch depending on the last bit (the less significant one) of the specified register. When it is zero, it changes to (or keeps in) ARM mode, and when it is 1 it changes to (or keeps in) THUMB mode. In other words, when a BX R_A or BLX R_A is executed, it will change to ARM mode if R_A = *offset*, and it will change to THUMB mode if R_A = *offset* + 1.

Stack instructions

We will first explain what the stack is. The **stack** can be defined as a 32-bit integers dynamic array that starts around RAM memory offset 0x027E0000 and is dynamically expanded to consecutive lower offsets. Note that this array elements have the same size as the registers: that is because this array is used for storing register values (when they are not going to be used in the current subroutine, or when there's not enough usable registers and some values have to be stored somewhere). The current pointer to the last element of the stack is stored in the SP register.

PUSH instruction stores the specified register values in the stack, while POP instruction loads back the values to registers. Both instructions modify the SP register value, as they add and delete elements from the array.



Keeping the SP register updated allows PUSH and POP instructions to know where they must operate in the RAM memory. Obviously, the SP register value is always a multiple of 4, because the elements are 4 bytes length.

A common usage of stack instructions happens when a BL instruction is executed: usually the first instruction of a called subroutine is a PUSH instruction that stores the LR (along with other registers). At the end of the subroutine, a POP is performed with the PC register (along with the other registers specified previously). This means that the LR value is stored in the stack, and later it ends in the PC register, so the program execution automatically jumps to the original LR value (that is, the instruction that was just after the BL).

Compilation fingerprints

There exist a lot of different compilation directives that convert the C source code to ASM code in quite different ways, with more or less optimization. In our case (in NDS games and in THUMB mode) we will find that a lot of C code structures have a specific ways to be converted in ARM instructions, allowing us to establish relationships between low-level programming language functions and assembly language subroutines.

The following list includes some examples of how C code is compiled into THUMB code.

Multiplying by a power of 2	
`u32 var1 = 7;` `var1 = var1*8;`	`MOV R0, #7` `LSL R0, R0, #3`
When a variable is multiplied by a constant that is a power of 2, the instruction **MUL** is not used. Instead it is used the **LSL** instruction, being value the exponent of 2 of the number that is multiplying the variable.	
Dividing by a power of 2	
`u32 var1 = 19;` `var1 = var1/16;`	`MOV R0, #19` `LSR R0, R0, #4`
When a variable is divided by a constant that is a power of 2, the instruction **LSR** is used, being value the exponent of 2 of the number that is dividing the variable.	
Dividing by an integer	

`u32 var1 = 46;`
`var1 = var1/6;`
	`MOV R0, #46`
	`MOV R1, #6`
	`BLX divisionSubroutine`

The NDS processor cannot make divisions in one instruction. That is why a subroutine needs to be called every time a division is performed, with R0 and R1 as arguments. A BLX is used because the division operand is a library function.	
Function output	
`u32 var1 = function() + 1;`	`BL function` `ADD R0, #1`
When a function is called and it returns an output, it is always stored in R0, with independence of its type (in some cases, a pointer to the variable is returned, because the output must fit in the 32-bit register).	

<table>
<tr><th colspan="2">Passing arguments to a function (1 to 4 arguments)</th></tr>
<tr><td><pre>function(2, 6, BULBASAUR, 0);</pre></td><td><pre>MOV R0, #2
MOV R1, #6
MOV R2, #1
MOV R3, #0
BL function</pre></td></tr>
<tr><td colspan="2">Arguments 1 to 4 are always stored in registers R0 to R3, so the called subroutine will then work with them.</td></tr>
<tr><th colspan="2">Passing arguments to a function (more than 4 arguments)</th></tr>
<tr><td><pre>function(2, 6, ARCEUS, 0, TRUE,
TRUE, 2, 0);</pre></td><td><pre>SUB SP, SP, #0x10
MOV R0, #1
STR R0, [SP]
STR R0, [SP, #4]
MOV R0, #2
STR R0, [SP, #8]
MOV R0, #0
STR R0, [SP, #0xC]
MOV R0, #2
MOV R1, #6
LDR R2, =0x1ED
MOV R3, #0
BL function
ADD SP, SP, #0x10</pre></td></tr>
<tr><td colspan="2">Arguments 1 to 4 are stored like we explained before, but further arguments are stored in the stack. First, the stack array is expanded in so many elements as arguments beyond 4 are passed to the function (in this case, the function takes 8 arguments, so 4 are passed by the register pathway and 4 by the stack pathway), so SP value needs to decrease for allowing these new 4 elements. Then, the stack-pathway arguments are stored in these new empty stack elements. Finally, registers R0 to R3 are loaded with the first 4 arguments. The called subroutine will later read the necessary arguments from the stack (with LDR RX, [SP, #X]). After the subroutine call, SP value must return to the original one.</td></tr>
<tr><th colspan="2">Immediate value load</th></tr>
<tr><td><pre>u32 var1 = 0x200;</pre></td><td><pre>MOV R0, #0x80
LSL R0, R0, #2</pre></td></tr>
<tr><td colspan="2">Instead of using LDR R0, =0x200 (it would spend 6 bytes: 2 bytes for the instruction and 4 bytes for the loaded value) a value under 256 is loaded and then multiplied by a power of 2. This only works if the desired value has a divisor that is a power of 2 and dividing the value by it results in a number lower than 256.</td></tr>
<tr><th colspan="2">Consecutive function calling</th></tr>
<tr><td><pre>u32 var1 = fnction1(fnction2(0));</pre></td><td><pre>MOV R0, #0
BL fnction2
BL fnction1</pre></td></tr>
<tr><td colspan="2">When the output of a function is the only argument of the following function, they are compiled consecutively. This is because fnction2 returns the output in R0, and fnction1 uses R0 as input (as explained before).</td></tr>
</table>

<table>
<tr><th colspan="2">Subroutine return when no subroutine is called inside it</th></tr>
<tr><td><pre>...
return;</pre></td><td><pre>...
BX LR</pre></td></tr>
<tr><td colspan="2">When a function does not call another function inside (in other words, the LR is not modified in any part of the subroutine) the return to the invoking subroutine is made jumping to the LR value with a BX instruction. In THUMB mode, the BL and BLX instructions automatically add +1 to the LR value, so it also returns to the previous processor mode. Remember that R0 must have the function's output, if it has any, before the BX instruction.</td></tr>
<tr><th colspan="2">Subroutine return when subroutines are called inside it</th></tr>
<tr><td><pre>...
return;</pre></td><td><pre>PUSH {R4-R7, LR}
...
POP {R4-R7, PC}</pre></td></tr>
<tr><td colspan="2">As explained before, LR is stored in the stack and later loaded directly into the PC register so the processor jumps to the invoking subroutine that had the BL or BLX. Usually registers R4 to R7 are also stored, or some of them, because they are used to store values in the invoking subroutine that should not be lost after the subroutine call. Remember that R0 must have the function's output, if it has any, before the POP instruction.</td></tr>
<tr><th colspan="2">Field access inside a structure</th></tr>
<tr><td><pre>typedef struct{
   u32 fld1;
   u16 fld2;
   u16 fld3;
}STRUCTURE

STRUCTURE var = function()
var.fld1 = var.fld2 + var.fld3;</pre></td><td><pre>BL function
LDRH R1, [R0, #4]
LDRH R2, [R0, #6]
ADD R1, R1, R2
STR R1, [R0, #0]</pre></td></tr>
<tr><td colspan="2">Structs will be covered in further documents. As we can see, function returns not the struct but the pointer to it (because a register cannot store a whole struct). Every field in the struct has a specific size, and accessing them in the RAM memory is done with LDR and STR instructions, knowing the relative pointer inside the struct.</td></tr>
<tr><th colspan="2">If</th></tr>
<tr><td><pre>if (var == 5) {
   function1();
} else {
   function2();
}
var++;</pre></td><td><pre>CMP R0, #5
BNE Else
BL function1
B Continue
Else:
BL function2
Continue:
ADD R0, #1</pre></td></tr>
<tr><td colspan="2">The If statements will be covered in further documents. They are compiled into CMP and branch instructions. Note that the assembly code always checks for the opposite condition.</td></tr>
</table>

For	

`u32 var = 0;`
`for (int i = 0; i < 4; i++) {`
`   var += i;`
`}`
`return var;`
	`MOV R0, #0`
	`MOV R1, #0`
	`Loop:`
	`ADD R0, R0, R1`
	`ADD R1, #1`
	`CMP R1, #4`
	`BLT Loop`
	`BX LR`

For loops will be covered in further documents. A register is always used as counter variable, and a branch with condition is at the end of the loop.	

What does IDA Pro?

We can split the disassembly tools in two types: the ones that interpret the code files (CrystalTile2, for example) and the ones that interpret the RAM memory. The last ones need the game to be running, but allow powerful methods for disassembling (breakpoints, tracing or checking the register values at any moment). These tools allow viewing the assembly code of the code files only when they are loaded in the RAM memory (so overlay files will only appear when they are needed).

IDA Pro allows to open code files without debugging (they must decompressed) but it works much better inspecting the RAM memory while the game is running.