23. Whirlwind Tour of ARM Assembly

23.1. Introduction

Very broadly speaking, you can divide programming languages into 4 classes. At the lowest level is machine code: raw numbers that the CPU decodes into instructions to execute. One step up is assembly. This is essentially machine code in words: each assembly instruction corresponds to one machine code instruction. Above this are compiled languages like C, which use structured language element to read more like English, but need to be compiled to machine code to be able to run. Finally, there are scripted languages like PHP (and usually VB and Java) which are run through interpreters configured to run the right kinds of machine code for the desired effects.

Every step up the ladder increases the human readability factor and portability, at the cost of runtime speed and program size. In the old days, programmers were Real Programmers and did their work in machine code or assembly because of clock speed and/or memory constraints. For PCs, these days are long gone and most work is done in the higher level languages. This, admittedly, is a good thing: code can be written faster and maintained more easily. However, there are still a few instances where the higher languages are insufficient. The GBA, with its 16.7Mhz CPU and less than 1 MB or work RAM is one of them. Here the inefficiency of the highest languages will cost you dearly, if it'd run at all. This is why most GBA work is done in C/C++, sometimes affectionately nicknamed ‘portable assembly’, because it still has the capability of working with memory directly. But sometimes even that isn't enough. Sometimes you *really* have to make every cycle count. And for this, you need **assembly**.

Now, in some circles the word “assembly” can be used to frighten small programmers. Because it is so closely tied to the CPU, you can make it do everything; but that also means you *have* to do everything. Being close to hardware also means you're bypassing all the safety features that higher languages may have, so that it's *much* easier to break things. So yeah, it is harder and more dangerous. Although some may prefer the term ‘adventurous’.

To program in assembly, you need to know how a processor actually works and write in a way it can understand, rather than rely on a compiler or interpreter to do it for you. There are no structured for- or while- loops or even if/else branches, just `goto`; no structs or classes with inheritance, and even datatypes are mostly absent. It's anarchy, but the lack of bureaucracy is exactly what makes fast code possible.

Speed/size issues aside, there are other reasons why learning assembly might be a good idea. Like I said, it forces you to actually *understand* how the CPU functions, and you can use that knowledge in your C code as well. A good example of this is the ‘best’ datatype for variables. Because the ARM processor is 32bit, it will prefer ints for most things, and other types will be slower, sometimes much slower. And while this is obvious from the description of the processor itself, knowledge of assembly will show you *why* they are slower.

A third reason, and not an inconsiderable one, is just for general coolness =B) . The very fact that it is harder than higher languages should appeal to your inner geek, who relishes such challenges. The simplicity of the statements themselves have an aesthetic quality as well: no messing about with classes, different loop styles, operator precedence, etc – it's one line, one opcode and never more than a handful of parameters.

Anyway, about this chapter. A complete document on assembly is nothing less than a full user's manual for a CPU. This would require an entire book in itself, which is not something I'm aiming at. My intention here is to give you an introduction (but a thorough one) to ARM assembly. I'll explain the most important instructions of the ARM and THUMB instruction sets, what you can and cannot do with them (and a little bit about why). I'll also cover how to use GCC's assembler to actually assemble the code and how to make your assembly and C files work together. Lastly, I'll give an example of a fast memory copier as an illustration of both ARM and Thumb code.

With that information, you should be able to do a lot of stuff, or at least know how to make use of the various reference documents out there. This chapter is not an island, I am assuming you have some or all of the following documents:

- The rather large official ARM7DTMI Technical manual (PDF): DDI0210B_7TDMI_R4.pdf.
- GBATek instruction reference: ARM / THUMB.

- Official ARM quick-references (PDF): ARM + Thumb
- Re-eject's quick-references (PDF): GAS / ARM / THUMB. (note: minor syntax discrepancies at times)
- GNU Assembler manual: GAS.

If you want more ARM/THUMB guides, you'll have to find them yourself.

23.2. General assembly

Assembly is little more than a glorified macro language for machine code. There is a one-to-one relationship between the assembly instructions and the actual machine code and assembly uses ***mnemonics*** for the operations the processor is capable of, which are much easier to remember than the raw binary. The tool that converts the asm code into machine code is the ***assembler***.

23.2.1. Basic operations

Every processor must be able to do basic data processing: arithmetic and bit manipulation. They should also have instructions to access memory, and be able to jump from one place in the code to another of conditionals and loops and such. However, different processors will have different ways of doing these things, and some operations of one set might not be present in another. For example, ARM lacks a division instruction, and can't perform data processing on memory directly. However, the ARM instruction set has some benefits too, like a fair amount of general-purpose registers and a simple instruction set, for the proper definition of “simple”. And it has a *very* nifty way of dealing with bit-shifts.

In the snippet below you can find a few examples of additions and memory reads in three different assembly languages: x86 (Intel), 68000, and ARM. The basic format is usually something like ‘`operation operand1, operand2, ...`’, though there are always exceptions. Note that where x86 and ARM put the destination in *Op1*, 68000 asm puts it in the last. The terminology of the registers is also different. Some semantics are pretty universal, the addition ‘`x += y` is found in all three, for example, but x86 also has a special instruction for increments by one, and in ARM the result register can be different from the two operands. These differences correspond to how the processors actually work! Higher languages allow you to use operations that do not seem present in the instruction set, but in fact they only *appear* to do so: the compiler/interpreter will convert it to a form the processor can actually handle.

Another point I must make here is that even for a given processor, there can be differences in how you write assembly. Assemblers aren't difficult to write, and there's nothing to stop you from using a different kind of syntax. Apart from the wrath of other programmers, of course.

```
// Some examples
// Addition and memory loads in different assemblies

// === x86 asm ========================================================
add     eax, #2         // Add immediate:   eax += 2;
add     eax, ebx        // Add register:    eax += ebx;
add     eax, [ebx]      // Add from memory: eax += ebx[0];
inc     eax             // Increment:       eax++;

mov     eax, DWORD PTR [ebx]        // Load int from memory:    eax= ebx[0];
mov     eax, DWORD PTR [ebx+4]      // Load next int:           eax= ebx[1];

// === 68000 asm ======================================================
ADD     #2, D0          // Add immediate:   D0 += 2;
ADD     D1, D0          // Add register:    D0 += D1;
ADD     (A0), D0        // Add from memory: D0 += A0[0];

MOVE.L  (A0), D0        // Load int from memory:    D0= A0[0];
MOVE.L  4(A0), D0       // Load next int:           D0= A0[1];

// === ARM asm ========================================================
add     r0, r0, #2      // Add immediate:   r0 += 2;
add     r0, r0, r1      // Add register:    r0 += r1;
add     r0, r1, r2      // Add registers:   r0= r1 + r2;

ldr     r0, [r2]        // Load int from memory:    r0= r2[0];
ldr     r0, [r2, #4]    // Load int from memory:    r0= r2[1];
ldmia   r2, {r0, r1}    // Load multiple:           r0= r2[0]; r1= r2[1];
```

23.2.2. Variables: registers, memory and the stack

In HLLs you have variables to work on, in assembly you can have registers, variables (that is, specific ranges in memory), and the stack. A ***register*** is essentially a variable inside the chip itself, and can be accessed quickly. The downside is that there are usually only a few of them, from just one to perhaps a few dozen. Most programs will require a lot more, which is why you can put variables in addressable memory as well. There's a lot more bytes in memory than in registers, but it'll also be slower to use. Note that both registers and memory are essentially **global** variables, change them in one function and you'll have changed them for the rest of the program. For local variables, you can use the stack.

The ***stack*** is a special region of memory used as, well, a stack: a Last-In, First-Out mechanism. There will be a special register called the ***stack pointer*** (SP for short) which contains the address of the top of the stack. You can *push* variables onto the top of the stack for safe keeping, and then *pop* them off once you're done with them, restoring the registers to their original values. The address of the stack (that is, the top of the stack, the contents of SP) is not fixed: it grows as you move deeper in the code's hierarchy, and shrinks as you move out again. The point is that each block of code should clean up after itself so that the stack pointer is the same before and after it. If not, prepare for a spectacular failure of the rest of the program.

For example, suppose you have functions `foo()` and which uses registers A, B, C and D. Function `foo()` calls function `bar()`, which also uses A, B and C, but in a different context than `foo()`. To make sure `foo()` would still work, `bar()` pushes A, B and C onto the stack at its start, then uses them the way it wants, and then pops them off the stack into A, B and C again when it ends. In pseudo code:

```
// Use of stack in pseudo-asm

// Function foo
foo:
    // Push A, B, C, D onto the stack, saving their original values
    push    {A, B, C, D}

    // Use A-D
    mov     A, #1           // A= 1
    mov     B, #2           // B= 2
    mov     C, #3           // well, you get the idea
    call    bar
    mov     D, global_var0

    // global_var1 = A+B+C+D
    add     A, B
    add     A, C
    add     A, D
    mov     global_var1, A

    // Pop A-D, restoring then to their original values
    pop     {A-D}
    return

// Function bar
bar:
    // push A-C: stack now holds 1, 2, 3 at the top
    push    {A-C}

    // A=2; B=5; C= A+B;
    mov     A, #2
    mov     B, #5
    mov     C, A
    add     C, B

    // global_var0= A+B+C (is 2*C)
    add     C, C
    mov     global_var, C

    // A=2, B=5, C=14 here, which would be bad when we
    // return to foo. So we restore A-C to original values.
    // In this case to: A=1, B=2, C=3
    pop     {A-C}
    return
```

While the syntax above is asm-like, it's not actually part of any assembly – at least not as far as I know. It is also particularly *bad* assembly, because it's inefficient in its use of registers, for one. If you were to write the corresponding C code and compile it (with optimizations, mind you), you get better code. But the point was here stack-use, not efficiency.

What you see here is that `foo()` sets A, B and C to 1, 2 and 3, respectively (`mov` stands for 'move', which usually comes down to assignment), and then calls `bar()`, which sets them to something else and sets a global variable called global_var0 to A+B+C. Because A, B and C are now different from what they were foo(), that function would use the wrong values in later calculations. To counter that, bar() uses the stack to save and restore A, B and C so that functions that call bar() still work. Note that foo() also uses the stack for A, B, C and D, because the function calling foo() may want to use them as well.

Stacking registers inside the called function is only a *guideline*, not a law. You could make the caller save/restore the variables that it uses. You could even not use the stack at all, as if you meant A, B and C to change and consider them return values of the function. By not setting the registers manually in bar(), A and B would effectively be function arguments. Or you could use the stack for function arguments. And return values. Or use both registers and the stack. The point is, you are free to do deal with them in any way you want. At least, in principle. In practice, there are guidelines written down by the original manufacturers, and while not written in stone, it can be considered bad form not to adhere to them. And you can see just *how* bad a form if you intend to make the code interface with compiled code, which *does* adhere to them.

23.2.3. Branching and condition codes

The normal operation for a computer is to take instructions one by one and execute them. A special register known as the ***program counter*** (PC) indicates the address of the next instruction. When it's time, the processor reads that instruction, does its magic and increments the program counter for the next instruction. This is a relatively straightforward process; things start to get interesting when you can set the program counter to a completely different address, redirecting the flow of the program. Some might say that you can really only speak of a *computer* if such a thing is possible.

The technical term for this redirection is ***branching***, though the term ‘jump’ is used as well. With branching you can create things like loops (infinite loops, mind you) and implement subroutines. The usual mnemonic for branching is something like b or j(mp)

```
// Asm version of the while(1) { ... } endless loop

// Label for (possible) branching destination
endless:

    ...         // stuff

    b endless // Branch to endless, for an endless loop.
```

The full power of branching comes from branching only when certain ***conditions*** are met. With that, you can perform if-else blocks and loops that can actually end. The conditions allowed depend on the processor, but the most common ones are:

- **Zero** (Z). If the result of operation was 0.
- **Negative** (N). Result was negative (i.e. most significant bit set).
- **Carry bit set** (C). If the ‘mostest’ significant bit is set (like bit 32 for 32bit operations).
- **Arithmetic overflow** (V). Like adding two positive numbers and getting a negative number because the result got too big for the registers.

These condition flags are stores in the ***Program Status Register*** (PSR), and each data processing instruction will set these one of more of these flags, depending on the outcome of the operation. Special versions of the branch instruction can use these flags to determine whether to make the jump.

Below you can see a simple example of a basic for-loop. The cmp instruction compares A to 16 and sets the PSR flags accordingly. The instruction bne stands for ‘branch if Not Equal’, which corresponds to a clear Z-flag. the reason for the Zero-flag's involvement is that the equality of two numbers is indicated by whether the difference between them is zero or not. So if there's a difference between A and 16, we jump back to for_start; if not, then we continue with the rest of the code.

```
// Asm version of for(A=0; A != 16; A++)

    mov     A, #0
// Start of for-loop.
for_start:

    ...             // stuff

    add     A, #1
    cmp     A, #16  // Compare A to 16
    bne for_start   // Branch to beginning of loop if A isn't 16
```

The number and names of the conditional codes depends on the platform. The ARM has 16 of these, but I'll cover these later.

23.2.4. An example: GCC generated ARM assembly

Before getting into ARM assembly itself, I'd like to show you a real-life example it. Assembly is an intermediary step of the build process, and you can capture GCC's assembly output by using the ‘-S’ or ‘-save-temps’ flags. This gives you the opportunity to see what the compiler is actually doing, to compare the C and assembly versions of a given algorithm, and provides quick pointers on how to code non-trivial things in assembly, like function calling, structures, loops etc. This section is optional, and you may not understand all the things here, but it is very educational nonetheless.

```
# Makefile settings for producing asm output
    $(CC) $(RCFLAGS) -S $<
```

```
// gen_asm.c :
//   plotting two horizontal lines using normal and inline functions.
#include <tonc.h>

void PlotPixel3(int x, int y, u16 clr)
{
    vid_mem[y*240+x]= clr;
}

int main()
{
    int ii;

    // --- using function ---
    ASM_CMT("using function");
    for(ii=0; ii<240; ii++)
        PlotPixel3(ii, 16, CLR_LIME);

    // --- using inline ---
    ASM_CMT("using inline");
    for(ii=0; ii<240; ii++)
        m3_plot(ii, 12, CLR_RED);

    while(1);

    return 0;
}
```

```
@@ gen_asm.s :
@@ Generated ASM (-O2 -mthumb -mthumb-interwork -S)
@@ Applied a little extra formatting and comments for easier reading.
@@ Standard comments use by @; my comments use @@

@@ Oh, and DON'T PANIC! :)

    .code   16
    .file   "gen_asm.c"     @@ - Source filename (not required)
    .text                   @@ - Code section (text -> ROM)

@@ <function block>
    .align  2               @@ - 2^n alignment (n=2)
    .global PlotPixel3      @@ - Symbol name for function
    .code   16              @@ - 16bit THUMB code (BOTH are required!)
    .thumb_func             @@ /
    .type   PlotPixel3, %function   @@ - symbol type (not req)
@@ Declaration : void PlotPixel3(int x, int y, u16 clr)
@@ Uses r0-r3 for params 0-3, and stack for param 4 and over
@@   r0: x
@@   r1: y
@@   r2: clr
PlotPixel3:
    lsl     r3, r1, #4      @@ \
    sub     r3, r3, r1      @@ - (y*16-1)*16 = y*240
    lsl     r3, r3, #4      @@ /
    add     r3, r3, r0      @@ - (y*240+x)
    mov     r1, #192        @@ - 192<<19 = 0600:0000
    lsl     r1, r1, #19     @@ /
    lsl     r3, r3, #1      @@ - *2 for halfword, not byte, offset
    add     r3, r3, r1
    @ lr needed for prologue
    strh    r2, [r3]        @@ store halfword at vid_mem[y*240+x]
    @ sp needed for prologue
    bx  lr
    .size   PlotPixel3, .-PlotPixel3    @@ - symbol size (not req)
@@ </ function block>

    .align  2
    .global main
    .code   16
    .thumb_func
    .type   main, %function
main:
    push    {r4, lr}            @@ Save regs r4, lr
    @ --- using function ---    @@ Comment from ASM_CMT, indicating
    .code   16                  @@   the PlotPixel3() loop
    mov     r4, #0              @@ r4: ii=0
.L4:
        mov     r2, #248        @@ - r2: clr= 248*4= 0x03E0= CLR_LIME
        lsl     r2, r2, #2      @@ /
        mov     r0, r4          @@ r0: x= ii
        mov     r1, #16         @@ r1: y= 16
        add     r4, r4, #1      @@ ii++
        bl      PlotPixel3      @@ Call PlotPixel3 (params in r0,r1,r2)
        cmp     r4, #240        @@ - loop while(ii<240)
        bne .L4                 @@ /
    @ --- using inline ---      @@ Comment from ASM_CMT, indicating
    .code   16                  @@   the m3_plot() loop
    ldr     r3, .L14            @@ r3: starting/current address (vid_mem[12*240])
    ldr     r2, .L14+4          @@ r2: terminating address (vid_mem[13*240])
    mov     r1, #31             @@ r1: clr (CLR_RED)
.L6:
        strh    r1, [r3]        @@ - *r3++ = clr
        add     r3, r3, #2      @@ /
        cmp     r3, r2          @@ - loop while(r3<r2)
        bne .L6                 @@ /
.L12:
    b   .L12
.L15:
    .align  2
.L14:
    .word   100669056       @@ 0600:1680 =&vid_mem[12*240]
    .word   100669536       @@ 0600:1886 =&vid_mem[13*240]
    .size   main, .-main

    .ident  "GCC: (GNU) 4.1.0 (devkitARM release 18)"
```

After the initial shock of seeing a non-trivial assembly file for the first time, you may be able to notice a few things, even without any prior knowledge of assembly.

- First, the assembly is much longer than the C file. This is not surprising as you can only have one instruction per line. While it makes the file longer, it also makes parsing each line easier.
- There are four basic types of line formats: labels (lines ending in a colon ':'), and instructions, and then in lines starting with a period or not. The instructions that start with a period are not really instructions, but ***directives***; they are hints to the assembler, not part of the CPU's instruction set. As such, you can expect them to differ between assemblers.

  The real instructions are usually composed of a mnemonic (`add`, `ldr`, `b`, `mov`) followed by register identifiers, numbers or labels. With a little thought, you should be able to piece together what each of these might do. For example, `add` performs an addition, `ldr` read something from memory, `b` branches, i.e. jumps to another memory address and `mov` does an assignment.

- Function structure and calling. In GAS, a function is preceded by a number of directives for alignment, code section and instruction set, and a ‘`.global`’ directive to make it globally visible. And a label to mark the start of the function of course. Note that for thumb functions, require a ‘`.thumb_func`’ directive as well as either ‘`.code 16`’ or ‘`.thumb`’. GCC also inserts a size info, but this are not required.

  Calling and returning from functions uses the `bl` and `bx` instructions. What isn't very clear from this code, except in my added comments, is that the arguments of the functions are put in registers r0-r3. What you definitely don't see is that if there are more than 4 parameters, these are put on the stack, and that the return value is put in r0.

  You *also* don't see that r0-r3 (and r12) are expected to be trashed in each function, so that the functions calling them should save their values if they want to use them after the call. The other registers (r4-r15) should be pushed into the stack by the called function. The standard procedure for function calling can be found in the AAPCS. Failure to adhere to this standard will break your code if you try to combine it with C or other asm.
- Loading the CLR_LIME color (0x03E0) doesn't happen in one go, but is spread over two instructions: a move and a shift. Why not move it in one go? Well, because it can't. The ARM architecture only allows byte-sized immediate values; bigger things have to be constructed in other ways. I'll get back to this later.
- The last thing I'd like to mention is the performance of the `PlotPixel3()` loop versus the `m3_plot()` loop, which you can find in the assembly because I've used a macro that can write asm comments in C. The `m3_plot()` loop contains 4 instructions. The `PlotPixel3()` loop takes 8, plus an additional 10 from the function itself. So that's 4 instructions against 18 instructions. The C code *seems* pretty much the same, so what gives?
  Welcome to the wonderful world of *function call overhead*. In principle, you only need the instructions of the shorter loop: a store, an add for the next destination, a compare and a loop-branch. Because `m3_plot()` is inlined, the compiler can see that this is all that's required and optimize the loop accordingly.

  In contrast, because `PlotPixel()` is a full function, the caller does not know what its internal code is, hence no optimizations are possible. The loop should reset the registers on *every iteration* because `PlotPixel()` will clobber them, making the loop in `main()` unnecessarily long. Furthermore, `PlotPixel3()` doesn't know under what conditions it will be called, so there are no optimizations there either. That means piecing together the destination in every iteration, rather than just incrementing it like the inline version does. All in all, you get a line plotter that's nearly 4 times as slow *purely* because you've used a function for a single line of code instead of inlining it via a macro or inline function. While anyone could have told you something like that would happen, actually looking at the differences leaves a stronger impression.

There is a lot more that could be learned from this code, but I'll leave it at this for now. The main aim was to show you what assembly (in this case THUMB asm) looks like. In this small piece of code, you can already see many of the elements that go into a full program. Even though the lack of variable identifiers is a bit of a pain, it should be possible to follow along with the code, just as you would with a C program. See, it's not all that bad now, is it?

On working by example

Looking at other people's code (in this case GCC's assembly) is a nice way of learning how to make things work, it is *not* a substitute for the manual. It may show you how to get something done, there is always the danger of getting them done wrongly or inefficiently. Programming is hardly ever trivial and you are likely to miss important details: the compiler may not be not optimising correctly, you could misinterpret the data, etc. This kind of learning often leads to cargo-cult programming, which often does more harm than good. If you want examples of these problems, look at nearly all other GBA tutorials and a lot of the available GBA demo code out there.

Assembling assembly

The assembler of the GNU toolchains is known as the GNU assembler or GAS, and the tool's name is `arm-none-eabi-as`. You can call this directly, or you can use the familiar `arm-none-eabi-gcc` to act as a gateway. The latter is probably a better choice, as it'll allow the use of the C preprocessor with ‘`-x assembler-with-cpp`’. That's right, you can then use macros, C-style comments *and* #include if you wish. A rule for assembling things might look something like this.

```
AS      := arm-none-eabi-gcc
ASFLAGS := -x assembler-with-cpp

# Rule for assembling .s -> .o files
$(SOBJ) : %.o : %.s
        $(AS) $(ASFLAGS) -c $< -o $@
```

This rule should work on the generated output of `gcc -S`. Note that it will probably not assemble under other assemblers (ARM SDT, Goldroad) because they have different standards for directives and comments and the like. I'll cover some important directives of GAS later, after we've seen what ARM assembly itself is like.

23.3. The ARM instruction set

The ARM core is a ***RISC*** (Reduced Instruction Set Computer) processor. Whereas CISC (Complex Instruction Set Computer) chips have a rich instruction set capable of doing complex things with a single instruction, RISC architectures try to go for more generalized instructions and efficiency. They have a comparatively large number of general-purpose registers and data instructions usually use three registers: one destination and two operands. The length of each instruction is the same, easing the decoding process, and RISC processors strive for 1-cycle instructions.

There are actually two instruction sets that the ARM core can use: ARM code with 32bit instructions, and a subset of this called THUMB, which has 16bit long instructions. Naturally, the ARM set is more powerful, but because the most used instructions can be found in both, an algorithm coded in THUMB uses less memory and may actually be faster if the memory buses are 16bit; which is true for GBA ROM and EWRAM and the reason why most of the code is compiled to THUMB. The focus in this section will be the ARM set, to learn THUMB is basically a matter of knowing which things you cannot do anymore.

The GBA processor's full name is ARM7TDMI, meaning it's an ARM 7 code (aka ARM v4), which can read **T**HUMB code, has a **D**ebug mode and a fast **M**ultiplier. This chapter has this processor in mind, but most of it should be applicable to other chips in the ARM family as well.

23.3.1. Basic features

ARM registers

ARM processors have 16 32bit registers named r0-r15, of which the last three are usually reserved for special purposes: **r13** is used as the stack pointer (SP); **r14** is the ***link register*** (LR), indicating where to return to from a function, and **r15** is the program counter (PC).. The rest are free, but there are a few conventions. The first four, **r0-r3**, are ***argument*** and/or ***scratch registers***; function parameters go here (or onto the stack), and these registers are expected to be clobbered by the called function. **r12** falls into this category too. The rest, **r4-r11**, are also known as ***variable registers***.

std	gcc	arm	description
r0-r3	r0-r3	a1-a4	argument / scratch
r4-r7	r4-r7	v1-v4	variable
r8	r8	v5	variable
r9	r9	v6/SB	platform specific
r10	sl	v7	variable
r11	fp	v8	variable / frame pointer
r12	ip	IP	Intra-Procedure-call scratch
r13	sp	SP	Stack Pointer
r14	lr	LR	Link Register
r15	pc	PC	Program Counter

Table 23.1. Standard and alternative register names.

ARM instructions

Nearly all of the possible instructions fall into the following three classes: ***data operations***, such as arithmetic and bit ops; ***memory operations***, load and store in many guises and ***branches*** for jumping around code for loops, ifs and function calls. The speed of instructions almost follows this scheme as well. Data instructions usually happen in a cycle; memory ops uses two or three and branches uses 3 or 4. The whole timing thing is actually a *lot* more complicated than this, but it's a useful rule of thumb.

All instructions are conditional

On most processors, you can only use branches conditionally, but on ARM systems you can attach the conditionals to *all* instructions. This can be very handy for small if/else blocks, or compound conditions, which would otherwise require the use of the more time-consuming branches. The code below contains asm versions of the familiar `max(a, b)` macro. The first one is the traditional version, which requires two labels, two jumps (although only one of those is executed) and the two instructions that actually do the work. The second version just uses two `mov`'s, but only one of them will actually be executed thanks to the conditionals. As a result, it is shorter, faster and more readable.

These kinds of conditionals shouldn't be used blindly, though. Even though you won't execute the instruction if the conditional fails, you still need to read it from memory, which costs one cycle. As a rough guideline, after about 3 skipped instructions, the branch would actually be faster.

```
@ // r2= max(r0, r1):
@ r2= r0>=r1 ? r0 : r1;

@ Traditional code
    cmp     r0, r1
    blt .Lbmax      @ r1>r0: jump to r1=higher code
    mov     r2, r0  @ r0 is higher
    b   .Lrest      @ skip r1=higher code
.Lbmax:
    mov     r2, r1  @ r1 is higher
.Lrest:
    ...             @ rest of code

@ With conditionals; much cleaner
    cmp     r0, r1
    movge   r2, r0  @ r0 is higher
    movlt   r2, r1  @ r1 is higher
    ...             @ rest of code
```

Another optional item is whether or not the status flags are set. Test instructions like `cmp` always set them, but most of the other require an ‘`-s`’ affix. For example, `sub` would not set the flags, but `subs` would. Because this kinda clashes with the plural ‘s’, I'm using adding an apostrophe for the plural form, so `subs` means `sub` with status flags, but `sub`'s means multiple `sub` instructions.

All instructions are conditional

Each instruction of the ARM set can be run conditionally, allowing shorter, cleaner and faster code.

The barrel shifter

A barrel shifter is a circuit dedicated to performing bit-shifts. Well, shifts and rotations, but I'll use the word ‘shift’ for both here. The barrel shifter is part of the ARM core itself and comes before any arithmetic so that you it can handle shifted numbers very fast. The real value of the barrel shifter comes from the fact that almost all instructions can apply a shift to one of their operands at no extra cost.

There are four barrel-shift operations: left shift (`lsl`), logical right-shift (`lsr`), arithmetic right-shift (`asr`) and rotate-right (`ror`). The difference between arithmetic and logical shift right is one of signed/unsigned numbers; see the bit ops section for details. These operations are attached to the last register in an operation, followed by an immediate value or a register. For example, instead of simply `Rm` you can have ‘`Rm, lsl #2`’ means `Rm<<2` and ‘`Rm, lsr Rs`’ for `Rm>>Rs`. Because shifted registers can apply to almost all instructions and I don't want to write it in full all the time, I will designate the shifted register as *Op2*.

Now this may seem like esoteric functionality, but it's actually very useful and more common than you think. One application is multiplications by 2^n±1, without resorting to relatively slow multiplication instructions. For example, x*9 is the same as x*(1+8) = x + x*8 = x+(x<<3). This can be done in a single `add`. Another use is in loading values from arrays, for which indices would have to be multiplied by the size of the elements to get the right addresses.

```
@ Multiplication by shifted add/sub

add r0, r1, r1, lsl #3      @ r0= r1+(r1<<3) = r1*9
rsb r0, r1, r1, lsl #4      @ r0= (r1<<2)-r1 = r1*15

@ word-array lookup: r1= address (see next section)
ldr r0, [r1, r2, lsl #2]    @ u32 *r1; r0= r1[r2]
```

Other uses are certainly possible as well. Like the conditional, you might not really need use of shifted `add`'s and such, but they allow for some *wonderfully* optimized code in the hands of the clever programmer. These are things that make assembly fun.

Shifts are free!

Or at least very nearly so. Shifts-by-value can be performed at no extra cost, and shifts-by-register cost only one cycle. Actually, bit-rotates behave like this as well, but they're rather rare and since I don't know of the correct term that encompasses both, I'll use the word “shift” for both.

Restricted use of immediate values

And now for one of the points that makes ARM assembly less fun. As I said, each instruction is 32bits long. Now, there are 16 condition codes, which take up 4 bits of the instruction. Then there's 2x four for the destination and first operand registers, one for the set-status flag, and then an assorted number of bits for other matters like the actual opcodes. The bottom line is that you only have 12 bits left for any immediate value you might like to use.

That's right 12. **A whole 12 bits**. You may have already figured out that, since this will only allow for 4096 distinct values, this presents a bit of a problem. And you'd be right. This is one of the major points of bad news for RISC processors: after assigning bits to instruction-type, registers and other fields, there's very little room for actual numbers left. So how is one to load a number like `0601:0000` (object VRAM) then? Well … you *can't*! At least, not in one go.

So, there is only a limited amount of numbers that can be used directly; the rest must be pieced together from multiple smaller numbers. Instead of just taking the 12 bits for a single integer, what the designers have done is split it into an 8bit number (n) and a 4bit rotation field (r). The barrel shifter will take care of the rest. The full immediate value v is given by:

(23.1) $$v = n \text{ ror } 2*r.$$

This means that you can create values like 255 (n=255, r=0) and 0x06000000 (n=6, r=4 (remember, rotate-*right*)). However, 511 and 0x06010000 are still invalid because the bit-patterns can't fit into one byte. For these invalid numbers you have two options: construct them in multiple instructions, or load them from memory. Both of these can become quite expensive so if it is possible to avoid them, do so.

The faster method of forming bigger numbers is a matter of debate. There are many factors involved: the number in mind, memory section, instruction set and amount of space left, all interacting in nasty ways. It's probably best not to worry about it too much, but as a guideline, I'd say if you can do it in two data instructions do so; if not, use a load. The easiest way of creating big numbers is with a special form of the `ldr` instruction: ‘`ldr Rd, =num`’ (note: no ‘#’!). The assembler will turn this into a mov if the number allows it, or an `ldr` if it doesn't. The space that the number needs will be created automatically as well.

```
    @ form 511(0x1FF) with mov's
    mov     r0, #256    @ 256= 1 ror 24, so still valid
    add     r0, #255    @ 256+255 = 511

    @ Load 511 from memory with special ldr
    @ NOTE: no '#' !
    ldr     r0,=511
```

That there is only room for an 8bit number + 4bit rotate for immediate operands is something you'll just have to learn to live with. If the assembler occasionally complains about invalid constants, you now know what it means and how you can correct for it. Oh, and if you thought this was bad, think of how it would work for THUMB code, which only has 16 bits to work with.

The only valid immediate values are rotated bytes

When instructions allow immediate values, the only permissible values are those that can be reduced to a byte rotated by an even number. 0xFF and 0x100 are allowed, but 0x1FF is not. This has consequences for data operations, but also for memory addressing since it will not be possible to load a full 32bit address in one go. You can either construct the larger value out of smaller parts, or use a load-assignment construct: ‘`ldr` *Rd*, =*num*’ which the assembler will convert into a `mov` if possible, or a PC-relative load if not.

Remember the previous note

Is this worth a separate note? Maybe not, but the previous note is important enough to remember. It is not exactly intuitive that code should behave that way and if you found yourself staring at the enigmatic invalid constant error message, you'd probably be lost without this bit of info.

23.3.2. Data instructions

The data operations carry out the calculations of a program, which includes both arithmetic and logical operations. You can find a summary of the data instructions in table 23.2. While this lists them in four groups, the only real division is between the multiplies and the rest. As you can see, there is **no** division instruction. While this can be considered highly annoying, as it turns out the need for division is actually quite small – small enough to cut it out of the instruction set, anyway.

Unlike some processors, ARM can only perform data processing on registers, not on memory variables directly. Most data instructions use one destination register and two operands. The first operand is always a register, the second can be four things: an immediate value or register (#*n* / `Rm`) or a register shifted by an immediate value or register (‘`Rm, lsl #`*n*’, ‘`Rm, lsl Rs`’, and similar for `lsr`, `asr` and `ror`). Because this arrangement is quite common, it is often referred to as simply *Op2*, even if it's not actually a second operand.

Like all instructions, data instructions can be executed conditionally by adding the appropriate affix. They can also alter the status flags by appending the `-s` prefix. When using both, the conditional affix always comes first.

opcode	operands	function
Arithmetic		
adc	Rd, Rn, Op2	Rd = Rn + Op2 + C
add	Rd, Rn, Op2	Rd = Rn + Op2
rsb	Rd, Rn, Op2	Rd = Op2 - Rn
rsc	Rd, Rn, Op2	Rd = Op2 - Rn - !C
sbc	Rd, Rn, Op2	Rd = Rn - Op2 - !C
sub	Rd, Rn, Op2	Rd = Rn - Op2
Logical ops		
and	Rd, Rn, Op2	Rd = Rn & Op2
bic	Rd, Rn, Op2	Rd = Rn &~ Op2
eor	Rd, Rn, Op2	Rd = Rn ^ Op2
mov	Rd, Op2	Rd = Op2
mvn	Rd, Op2	Rd = ~Op2
orr	Rd, Rn, Op2	Rd = Rn | Op2

opcode	operands	function
Status ops		
cmp	Rn, Op2	Rn - Op2
cmn	Rn, Op2	Rn + Op2
teq	Rn, Op2	Rn ^ Op2
tst	Rn, Op2	Rn & Op2
Multiplies		
mla	Rd, Rm, Rs, Rn	Rd = Rm * Rs + Rn
mul	Rd, Rm, Rs	Rd = Rm * Rs
smlal	RdLo, RdHi, Rm, Rs	RdHiLo += Rm * Rs
smull	RdLo, RdHi, Rm, Rs	RdHiLo = Rm * Rs
umlal	RdLo, RdHi, Rm, Rs	RdHiLo += Rm * Rs
umull	RdLo, RdHi, Rm, Rs	RdHiLo = Rm * Rs

23.2: Data processing instructions. Basic format `op{cond}{s} Rd, Rn, Op2`, `cond` and `s` are the optional condition and status codes, and *Op2* a shifted register.

The first group, arithmetic, only contains variants of addition and subtraction. `add` and `sub` are their base forms. `rsb` is a special thing that reverses the operand order; the difference with the regular `sub` is that *Op2* is now the *minuend* (the thing subtracted from). Only *Op2* is allowed to have immediate values and shifted registers, which allows you to negate values ($0-x$) and fast-multiply by 2^n-1.

The variants ending in ‘`c`’ are additions and subtractions with carry, which allows for arithmetic for values larger than the register size. For example, consider you have 8bit registers and want to add 0x00FF and 0x0104. Because the latter doesn't fit into one register, you have to split it and then add twice, starting with with the least significant byte. The lowest byte addition gives 0xFF+0x04=0x103, represented by 0x03 in the destination register and a set carry flag. For the second part you have to add 0x00 and 0x01 from the operands, *and* the carry from the lower byte, giving 0x00+0x01+1 = 0x02. Now string the separate parts together to give 0x0203.

Because ARM registers are 32bit wide you probably won't be using the those instructions much, but you never know.

The second group are the bit operations, most of which you should be familiar with already. The all have exact matches in C operators, with the exception of bit-clear. However, the value of such an instruction should be obvious. You will notice a distinct absence of shift instructions here, for the simple reason that they're not really necessary: thanks to the barrel shifter, the `mov` instruction can be used for shifts and rotates. ‘r1 = r0<<4’ could be written as ‘`mov r1, r0, lsl #4`’.

I have mentioned this a couple of times now, but as we're dealing with another language now it bares repeating: there is a difference between right-shifting a signed and unsigned numbers. Right-shifts remove bits from the top; unsigned numbers should be zero-extended (filled with 0), but signed numbers should be sign-extended (filled with the original MSB). This is the difference between a ***logical shift right*** and an ***arithmetic shift right***. This doesn't apply to left-shifts, because that fills zeroes either way.

The third group isn't much of a group, really. The status flag operations set the status bits according to the results of their functionality. Now, you can do this with the regular instructions as well; for example, a compare (`cmp`) is basically a subtraction that also sets the status flags, i.e., a `subs`. The only real difference is that this time there's no register to hold the result of the operation.

Lastly, the multiplication formats. At the table indicates, you cannot use immediate values; if you want to multiply with a constant you *must* load it into a register first. Second, no *Op2* means no shifted registers. There is also a third rule that Rd and Rm can't use the same register, because of how the multiplication algorithm is implemented. That said, there don't *seem* to be any adverse effects using Rd=Rm.

The instruction `mla` stands for ‘multiply with accumulate’, which can be handy for dot-products and the like. The `mull` and `mlal` instructions are for 64bit arithmetic, useful for when you expect the result not to fit into 32bit registers.

```
@ Possible variations of data instructions
add     r0, r1, #1          @ r0 = r1 + 1
add     r0, r1, r2          @ r0 = r1 + r2
add     r0, r1, r2, lsl #4  @ r0 = r1 + r2<<4
add     r0, r1, r2, lsl r3  @ r0 = r1 + r2<<r3

@ op= variants
add     r0, r0, #2          @ r0 += 2;
add     r0, #2              @ r0 += 2; alternative  (but not on all assemblers)

@ Multiplication via shifted add/sub
add     r0, r1, r1, lsl #4  @ r0 = r1 + 16*r1 = 17*r1
rsb     r0, r1, r1, lsl #4  @ r0 = 16*r1 - r1 = 15*r1
rsb     r0, r1, #0          @ r0 =     0 - r1 = -r1

@ Difference between asr and lsr
mvn     r1, #0              @ r1 = ~0 = 0xFFFFFFFF = -1
mov     r0, r1, asr #16     @ r0 = -1>>16 = -1
mov     r0, r1, lsr #16     @ r0 = 0xFFFFFFFF>>16 = 0xFFFF = 65535


@ Signed division using shifts. r1= r0/16
@ if(r0<0)
@     r0 += 0x0F;
@  r1= r0>>4;
mov     r1, r0, asr #31         @ r0= (r0>=0 ? 0 : -1);
add     r0, r0, r1, lsr #28     @ += 0 or += (0xFFFFFFFF>>28 = 0xF)
mov     r1, r0, asr #4          @ r1 = r0>>4;
```

23.3.3. Memory instructions: load and store

Because ARM processors can only perform data processing on registers, interactions with memory only come in two flavors: loading values from memory into registers and storing values into memory from registers.

The basic instructions for that are `ldr` (LoaD Register) and `str` (STore Register), which load and store words. Again, the most general form uses two registers and an *Op2*:

op{cond}{type} Rd, [Rn, *Op2*]

Here *op* is either `ldr` or `str`. Because they're so similar in appearance, I will just use `ldr` for the remainder of the discussion on syntax, except when things are different. The condition flag again goes directly behind the base opcode. The *type* refers to the datatype to load (or store), which can be words, halfwords or bytes. The word forms do not use any extension, halfwords use `-h` or `-sh`, and bytes use `-b` and `-sb`. The extra `s` is to indicate a signed byte or halfword. Because the registers are 32bit, the top bits need to be sign-extended or zero-extended, depending on the desired datatype.

The first register here, `Rd` can be either the destination or source register. The thing between brackets always denotes the memory address; `ldr` means load *from* memory, in which case `Rd` is the destination, and `str` means store *to* memory, so `Rd` would be the source there. `Rn` is known as the ***base register***, for reasons that we will go into later, and *Op2* often serves as an offset. The combination works very much like array indexing and pointer arithmetic.

Memory ops vs C pointers/arrays

To make the comparison to C a little easier, I will sometimes indicate what happens using pointers, but in order to do that I will have to indicate the type of the pointer somehow. I could use some horrid casting notation, but it would be easiest to use a form of arrays for this, and use the register-name + an affix to show the data type. I'll use ‘_w’ for words, ‘_h’ for halfwords, and ‘_b’ for bytes, and ‘_sw’, etc. for their signed versions. For example, `r0_sh` would indicate that `r0` is a signed halfword pointer. This is just a useful bit of shorthand, not actually part of assembly itself.

```
@ Basic load/store examples. Assume r1 contains a word-aligned address
ldr     r0, [r1]    @ r0= *(u32*)r1; //or r0= r1_w[0];
str     r0, [r1]    @ *(u32*)r1= r0; //or r1_w[1]= r0;
```

Addressing modes

There are several ways of interacting, known as ***addressing modes***. The simplest form is *direct addressing*, where you indicate the address directly via an immediate value. However, that mode is unavailable to ARM systems because full addresses don't fit in the instruction. What we do have is several indirect addressing forms.

The first available form is ***register indirect addressing***, which gets the address from a register, like ‘`ldr Rd, [Rn]`’. An extension of this is ***pre-indexed addressing***, which adds an offset to the base register before the load. The base form of this is ‘`ldr Rd, [Rn,` *`Op2`*`]`’. This is very much like array accesses. For example ‘`ldr r1, [r0, r2, lsl #2]`’ corresponds to `r0_w[r2]`: an word-array load using `r2` as the index.

Another special form of this is ***PC-relative addressing***, which makes up for not having direct addressing. Suppose you have a variable in memory somewhere. While you may not be able to use that variable's address directly, what you can do is store the address close to where you are in the code. *That* address is at a certain allowed offset from the program counter register (PC), so you could load the variable's address from there and then read the variable's contents. You can also use this to load constants that are too large to fit into a shifted byte.

While it is possible to calculate the required offset manually, you'll be glad to know you can let the assembler do this for you. There are two ways of doing this. The first is to create a ***data-pool*** where you intend to put the addresses and constants, and label it. You can then get its address via ‘`ldr Rd,` *`LabelName`*’ (note the absence of brackets here). The assembler will turn this into pc-relative loads. The second method is to let the assembler do all the work by using ‘`ldr Rd, =`*`foo`*’, where *foo* is the variable name or an immediate value. The assembler will then allocate space for *foo* itself. Please remember that using =*varname* does **not** load the variable itself, only its address.

And then there are the so-called ***write-back*** modes. In the pre-index mode, the final address was made up of `Rn`+*Op2*, but that had no effect on `Rn`. With write-back, the final address is put in `Rn`. This can be useful for walking through arrays because you won't need an actual index.

There are two forms of write-back, pre-indexing and post-indexing. Pre-indexing write-back works much like the normal write-back and is indicated by an exclamation mark after the brackets: ‘`ldr Rd, [Rn,` *`Op2`*`]!`’. Post-indexing doesn't add *Op2* to the address (and `Rn`) until *after* the memory access; its format is ‘`ldr Rd, [Rn],` *`Op2`*’.

```
@ Examples of addressing modes
@ NOTE: *(u32*)(address+ofs) is the same as ((u32*)address)[ofs/4]
@   That's just how array/pointer offsets work
    mov     r1, #4
    mov     r2, #1
    adr     r0, fooData     @ u32 *src= fooData;
@ PC-relative and indirect addressing
    ldr     r3, fooData             @ r3= fooData[0];   // PC-relative
    ldr     r3, [r0]                @ r3= src[0];       // Indirect addressing
    ldr     r3, fooData+4           @ r3= fooData[1];   // PC-relative
    ldr     r3, [r0, r1]            @ r3= src[1];       // Pre-indexing
    ldr     r3, [r0, r2, lsl #2]    @ r3= src[1]        // Pre-index, via r2
@ Pre- and post-indexing write-back
    ldr     r3, [r0, #4]!           @ src++;    r3= *src;
    ldr     r3, [r0], #4            @ r3= *src; src++;
@ u32 fooData[3]= { 0xF000, 0xF001, 0xF002 };
fooData:
    .word   0x0000F000
    .word   0x0000F001
    .word   0x0000F002
```

PC-relative specials

PC-relative instructions are common, and have a special shorthand that is easier and shorter to use than creating a pool of data to load from. The format for this is ‘`ldr Rd, =`*foo*’, where *foo* is a label or an immediate value. In both cases, a pool is created to hold these numbers automatically. Note that the value for the label is its *address*, not the address' contents.

If the label is near enough you can also use `adr`, which is assembled to a PC-add instruction. This will not create a pool-entry.

```
@ Normal pc-relative method:
@   create a nearby pool and load from it
    ldr     r0, .Lpool      @ Load a value
    ldr     r0, .Lpool+4    @ Load far_var's address
    ldr     r0, [r0]        @ Load far_var's contents
.Lpool:
    .word   0x06010000
    .word   far_var
```

```
@ Shorthand: use ldr= and GCC will manage the pool for you
    ldr     r0,=0x06010000  @ Load a value
    ldr     r0,=far_var     @ Load far_var's address
    ldr     r0, [r0]        @ Load far_var's contents
```

Note that I'm not actually creating `far_var` here; just storage room for its address. Creation of variables is covered later.

Data types

It is also possible to load/store bytes and halfwords. The opcodes for loads are `ldrb` and `ldrh` for unsigned, and `ldrsb` and `ldrsh` for signed bytes and halfwords, respectively. The ‘r’ in the signed versions is actually optional, so you'll also see `ldsb` and `ldsh` now and then. As stores can cast away the more significant bytes anyway, `strb` and `strh` will work for both signed and unsigned stores..

All the things you can do with `ldr/str`, you can do with the byte and halfword versions as well: PC-relative, indirect, pre/post-indexing it's all there … with one exception. The signed-byte load (`ldsb`) and *all* of the halfword loads and stores cannot do shifted register-loads. Only `ldrb` has the complete functionality of the word instructions. The consequence is that signed-byte or halfword arrays may require extra instructions to keep the offset and index in check.

Oh, one more thing: alignment. In C, you could rely on the compiler to align variables to their preferred boundaries. Now that you're taking over from the compiler, it stands to reason that you're also in charge of alignment. This can be done with the ‘.align *n*’ directive, with aligns the next piece of code or data to a 2^n boundary. Actually, you're supposed to properly align code as well, something I'm taking for granted in these snippets because it makes things easier.

```
    mov     r2, #1
@ Byte loads
    adr     r0, bytes
    ldrb    r3, bytes       @ r3= bytes[0];     // r3= 0x000000FF= 255
    ldrsb   r3, bytes       @ r3= (s8)bytes[0]; // r3= 0xFFFFFFFF= -1
    ldrb    r3, [r0], r2    @ r3= *r0_b++;      // r3= 255, r0++;
@ Halfword loads
    adr     r0, hwords
    ldrh    r3, hwords+2    @ r3= words[1];     // r3= 0x0000FFFF= 65535
    ldrsh   r3, [r0, #2]    @ r3= (s16)r0_h[1]; // r3= 0xFFFFFFFF= -1
    ldrh    r3, [r0, r2, lsl #1]    @ r3= r0_h[1]? No! Illegal instruction :(

@ Byte array: u8 bytes[3]= { 0xFF, 1, 2 };
bytes:
    .byte   0xFF, 1, 2
@ Halfword array u16 hwords[3]= { 0xF001, 0xFFFF, 0xF112 };
    .align  1     @ align to even bytes REQUIRED!!!
hwords:
    .hword  0xF110, 0xFFFF, 0xF112
```

Block transfers

Block transfers allow you to load or store multiple successive words into registers in one instruction. This is useful because it saves on instructions, but more importantly, it saves time because individual memory instructions are quite costly and with block transfers you only have to pay the overhead once. The basic instructions for block transfers are `ldm` (LoaD Multiple) and `stm` (STore Multiple), and the operands are a base register (with an optional exclamation mark for `Rd` write-back) and a list of registers between braces.

op{cond}{mode} Rd{!}, {*Rlist*}

This register list can be comma separated, or hyphenated to indicate a range. For example, `{r4-r7, lr}` means registers r4, r5, r6, r7 and r14. The order in which the registers are actually loaded or stored are **not** based on the order in which they are specified in this list! Rather, the list indicates the number of words used (in this case 5), and the order of addresses follows the index of the registers: the lowest register uses the lowest address in the block, etc.

The block-transfer opcodes can take a number of affixes that determine how the block extends from the base register `Rd`. The four possibilities are: `-IA`/`-IB` (Increment After/Before) and `-DA`/`-DB` (Decrement After/Before). The differences are essentially those between pre/post-indexing and incrementing or decrementing from the base address. It should be noted that these increments/decrements happen regardless of whether the base register carries an exclamation mark or not: that thing only indicates that the base register *itself* is updated afterwards.

```
    adr     r0, words+16    @ u32 *src= &words[4];
                            @              r4, r5, r6, r7
    ldmia   r0, {r4-r7}     @ *src++     :  0,  1,  2,  3
    ldmib   r0, {r4-r7}     @ *++src     :  1,  2,  3,  4
    ldmda   r0, {r4-r7}     @ *src--     : -3, -2, -1,  0
    ldmdb   r0, {r4-r7}     @ *--src     : -4, -3, -2, -1
    .align  2
words:
    .word   -4, -3, -2, -1
    .word    0,  1,  2,  3, 4
```

The block transfers are also used for stack-work. There are four types of stacks, depending on whether the address that `sp` points to already has a stacked value or not (Full or Empty), and whether the stack grows down or up in memory (Descending/Ascending). These have special affixes (`-FD`, `-FA`, `-ED` and `-EA`) because using the standard affixes would be awkward. For example, the GBA uses an FD-type stack, which means that pushing is done with `stmdb` because decrementing after the store would overwrite an already stacked value (full stack), but popping requires `ldmia` for similar reasons. A `stmfd/ldmfd` pair is much easier to deal with. Or you could just use `push` and `pop`, which expand to '`stmfd sp!,`' and '`ldmfd sp!,`', respectively.

Block op	Standard	Stack alt
Increment After	ldmia / stmia	ldmfd / stmea
Increment Before	ldmib / stmib	ldmed / stmfa
Decrement After	ldmda / stmda	ldmfa / stmed
Decrement Before	ldmdb / stmdb	ldmea / stmfd

Table 23.3: Block transfer instructions.

push and pop are not universal ARM instructions

They seem to work for devkitARM r15 and up (haven't checked older versions), but DevKitAdv for example doesn't accept them. Just try and see what happens.

23.3.4. Conditionals and branches

Higher languages typically have numerous methods for implementing choices, loops and function calls. The all come down to the same thing though: the ability to move the program counter and thereby diverting the flow of the program. This procedure is known as branching.

There are three branching instructions in ARM: the simple branch `b` for ifs and loops, the branch with link `bl` used for function calls, and branch with exchange `bx` used for switching between ARM and THUMB code, returning from functions and out-of-section jumps. `b` and `bl` use a label as their argument, but `bx` uses a register with the address to branch to. Technically there are more ways of branching (PC is just another register, after all) but these three are the main ones.

Status flags and condition codes

I've already mentioned part of this in the introduction, so I'll make this brief. The ARM processor has 4 status flags, (**Z**)ero, (**N**)egative, (**C**)arry and signed o(**V**)erflow, which can be found in the program status register. There are actually two of these: one for the *current* status (CPSR) and a *saved* status register (SPSR), which is used in interrupt handlers. You won't have to deal with either of these, though, as reacting to status registers usually goes through the conditional codes (table 23.4). But first, a few words about the flags themselves:

- **Zero** (Z). If the result of operation was 0.
- **Negative** (N). Result was negative (i.e. most significant bit set).
- **Carry bit set** (C). If the ‘mostest' significant bit is set (like bit 32 for 32bit operations).
- **Arithmetic overflow** (V). Like adding two positive numbers and getting a negative number because the result got too big for the registers.

Each of the data instructions can set the status flags by appending `-s` to the instruction, except for `cmp`, `cmn`, `tst` and `teq`, which always set the flags.

Table 23.4 lists 16 affixes that can be added to the basic branch instruction. For example, `bne Label` would jump to `Label` if the status is non-zero, and continue with the next instruction if it isn't.

Affix	Flags	Description
eq	Z=1	Zero (EQual to 0)
ne	Z=0	Not zero (Not Equal to 0)
cs / hs	C=1	Carry Set / unsigned Higher or Same
cc / lo	C=0	Carry Clear / unsigned LOwer
mi	N=1	Negative (MInus)
pl	N=0	Positive or zero (PLus)
vs	V=1	Sign overflow (oVerflow Set)
vc	V=0	No sign overflow (oVerflow Clear)
hi	C=1 & Z=0	Unsigned HIgher
ls	C=0 | Z=1	Unsigned Lower or Same
ge	N=V	Signed Greater or Equal
lt	N != V	Signed Less Than
gt	Z=0 & N=V	Signed Greater Than
le	Z=1 | N != V	Signed Less or Equal
al	-	ALways (default)
nv	-	NeVer

Table 23.4: conditional affixes.

To use these condition codes properly, you need to know what each stands for, but also how the data operations set the flags. The effect on the status flags depends on the instruction itself, and not all flags are affected by all instructions. For example, overflow only has meaning for arithmetic, not bit operations.

In the case of Z and N, the case is pretty easy. The operation gives a certain 32bit value as its result; if it's 0, then the Zero flag is set. Because of two's complement, the Negative flag is the same as bit 31. The reason `-eq` and `ne` are linked to the zero flags is because a comparison (`cmp`) is basically a subtraction: it looks at the difference between the two numbers and when that's zero, then the numbers are equal.

For the carry bit it can get a little harder. The best way to see it is as an extra most significant bit. You can see how this work in the example of table 23.5. Here we add two unsigned numbers, 2^{31} = 0x80000000. When adding them, the result would overflow 32bits, giving 0 and not 2^{32}. However, that overflowed bit will go into the carry. With the `adc` instruction you could then go on to build adders for numbers larger thatn the registers.

2^{31}	`8000 0000`
2^{31}	`8000 0000 +`
2^{32}	`1 0000 0000`

Table 23.5: carry bit in (unsigned) addition.

Bit-operations like `orr` or `and` don't affect it because they operate purely on the lower 32bits. Shifts, however do.

You may find it odd that `-cc` is the code for unsigned higher than. As mentioned, a comparison is essentially a subtraction, but when you subtract, say 7−1, there doesn't really seem to be a carry here. The key here is that subtractions are infact forms of additions: 7−1 is actually 7+0xFFFFFFFF, which would cause an overflow into the carry bit. You can also thing of subtractions as starting out with the carry bit set.

The overflow flag indicates *signed* overflow (the carry bit would be unsigned overflow). Note, this is *not* merely a sign change, but a sign change the wrong way. For example, an addition of two positive numbers *should* always be positive, but if the numbers are big enough (say, 2^{30}, see table 23.6) then the results of the lower 30 bits may overflow into bit 31, therefore changing the sign and you'll have an incorrect addition. For subtraction, there can be a similar problem. Short of doing the full operation and checking whether the signs are correct, there isn't a simple way of figuring out what counts as overflow, but fortunately you don't have to. Usually overflow is only important for signed comparisons, and the condition mnemonics themselves should provide you with enough information to pick the right one.

$+2^{30}$	4000 0000
$+2^{30}$	4000 0000 +
-2^{31}	8000 0000

Table 23.6: sign overflow.

With these points in mind, the conditional codes shouldn't be too hard to understand. The descriptions tell you what code you should use when. Also, don't forget that any instruction can be conditionally executed, not just a branch.

The basic branch

Let's start with the most basic of branches, `b`. This is the most used branch, used to implement normal conditional code and loops of all kinds. It is most often used in conjunction with one of the 16 conditional codes of table 23.4. Most of the times a branch will look something like this:

```
@ Branch example, pseudo code
    data-ops, Rd, Rn, Op2    @ Data operation to set the flags
    bcnd-code .Llabel        @ Branch upon certain conditions

    @ more code A

.Llabel:                     @ Branch goes here
    @ more code B
```

First, you have a data processing instruction that sets the status flags, usually a `subs` or `cmp`, but it can be any one of them. Then a b*cond* diverts the flow to `.Llabel` if the conditions are met. A simple example of this would be a division routine which checks if the denominator is zero first. For example, the `Div()` routine that uses BIOS Call #6 could be safeguarded against division by 0 like this:

```
@ int DivSafe(int num, int den);
@ \param num    Numerator (in r0)
@ \param den    Denominator (in r1)
@ \return       r0= r0/r1, or INT_MAX/INT_MIN if r1 == 0
DivSafe:
    cmp     r1, #0
    beq     .Ldiv_bad    @ Branch on r1 == 0
    swi     0x060000
    bx      lr
.Ldiv_bad:
    mvn     r1, #0x80000000     @ \
    sub     r0, r1, r0, asr #31 @ - r0= r0>=0 ? INT_MAX : INT_MIN;
    bx      lr
```

The numerator and denominator will be in registers r0 and r1, respectively. The `cmp` checks whether the denominator is zero. If it's not, no branch is taken, the swi 6 is executed and the function returns afterwards. If it is zero, the `beq` will take the code to `.Ldiv_bad`. The two instructions there set r0 to either INT_MAX (2^{31}−1 = 0x7FFFFFFF) or INT_MIN (−2^{31} = 0x80000000), depending on whether r0 is positive or negative. If it's a little hard to see that, `mvn` inverts bits, so the first line after `.Ldiv_bad` sets r0 to INT_MAX. The second line we've seen before: '`r0, asr #31`' does a sign-extension in to all other bits, giving 0 or −1 for positive and negative numbers, respectively, giving INT_MAX− −1 = INT_MIN for negative values of r0. Little optimizing tricks like these decide if you're fit to be an assembly programmer; if not you could just as well let the compiler do them, because it does know. (It's where I got the '`asr #31`' thing from in the first place.)

Now in this case I used a branch, but in truth, it wasn't even necessary. The non-branch part consists of one instruction, and the branched part of two, so using conditional instructions throughout would have been both shorter and faster:

```
@ Second version using conditionally executed code
DivSafe:
    cmp     r1, #0
    mvneq   r1, #0x80000000
    subeq   r0, r1, r0, asr #31
    swine   0x060000
    bx      lr
```

If the denominator is not zero, the `mvneq` and `subeq` are essentially skipped. Actually, not so much skipped, but turned into `nop`: non-operations. So is `swine` (i.e., `swi` + `ne`, no piggies here) if it is zero. True, the division line has increased by a cycle, not taking the branch makes the exception line a little faster and the function itself has shrunk from 7 to 5 instructions.

Symbol vs internal labels

In the first `DivSafe` snippet, the internal branch destination used a `.L` prefix, while the function label did not. The `.L` prefix is used by GCC to indicate labels for the sake of labels, as opposed to symbol labels like `DivSafe`. While not required, it's a useful convention.

Major and minor branches

Any sort of branch will create a fork in the road and, depending on the conditions, one road will be taken more often. That would be the ***major*** branch. The other one would be the ***minor*** branch, probably some sort of exception. The branch instruction, `b`, represents a deviation from the normal road and is relatively costly, therefore it pays to have to branch to the exceptions. Consider these possibilities:

```
// Basic if statement in C
if(r0 == 0)
{   /* IF clause */   }
...
```

```
@ === asm-if v1 : 'bus stop' branch ===
    cmp     r0, #0
    beq     .Lif
.Lrest:
    ...
    bx      lr      @ function ends
.Lif
    @ IF clause
    b   .Lrest


@ === asm-if v2 : 'skip' branch ===
    cmp     r0, #0
    bne     .Lrest
    @ IF clause
.Lrest:
    ...
    bx      lr      @ function ends
```

The first version is more like the C version: it splits off for the IF-clause and the returns to the rest of the code. The flow branches twice if the conditions are met, but if they aren't the rest of the code doesn't branch at all. The second version branches if the conditions *aren't* met: it skips the IF-clause. Overall, the assembly code is simpler and shorter, but the fact that the branch-conditions are inverted with respect to the C version could take some getting used to.

So which to use? Well, that depends actually. All things being equal, the second one is better because it's one instruction and label shorter. As far as I know, this is what GCC uses. The problem is that some things may be more equal than others. If the IF-clause is exceptional (i.e., the minor branch), it'd mean that the second version almost always takes the branch, while the first version would hardly ever branch, so on average the latter would be faster.

Which one you chose is entirely up to you as you know your intentions best. I hope. For the remainder of this chapter I'll use the skip-branch because in demonstrations things usually are equal. Of course, if the clause is small enough you can just use conditional instructions and be done with it `:)`.

Common branching constructs

Even though all you have now is `b`, it doesn't mean you can't implement branching construct found in HLLs in assembly. After all, the compiler seems to manage. Here's a couple of them.

if-elseif

The `if-elseif` is an extension of the normal `if-else`, and from it you can extend to longer `if-elseif-else` chains. In this case I want to look at a wrapping algorithm for keeping numbers within certain boundaries: the number *x* should stay within range [*mn*, *mx*⟩, if it exceeds either boundary it should come out the other end. In C it looks like this:

```
// wrap(int x, int mn, int mx), C version:
int res;
if(x >= mx)
    res= mn + x-mx;
else if(x < mn)
    res= mx + x-mn;
else
    res= x;
```

The straightforward compilation would be:

```
@ r0= x ; r1= mn ; r2= mx
    cmp     r0, r2
    blt     .Lx_lt_mx       @ if( x >= mx )
    add     r3, r0, r1      @   r0= mn + x-mx
    sub     r0, r3, r2
    b       .Lend
.Lx_lt_mx:
    cmp     r0, r1          @
    bge     .Lend           @ if( x < mn )
    add     r3, r0, r2      @   r0= mx + x-mn;
    sub     r0, r3, r1
.Lend:
    ...
```

This is what GCC gives, and it's pretty good. The ordering of the clauses remained, which means that the condition for the branches have to be inverted, so ‘`if(x >= mx) {}`’ becomes ‘skip if NOT *x* >= *mx*’. At the end of each clause, you'd need to skip all the others: branch to `.Lend`. The conditional branches mean ‘go to the next branch’.

And now an optimized version. First, a `cmp` is equivalent to `sub` except that it doesn't put the result in a register. However, as we need the result later on anyway, we might as well combine the ‘`cmp`’ and ‘`sub`’. Secondly, the clauses are pretty small, so we can use conditional ops as well. The new version would be:

```
@ Optimized wrapper
    subs    r3, r0, r2      @ r3= x-mx
    addge   r0, r3, r1      @   x= x-mx + mn
    bge     .Lend
    subs    r3, r0, r1      @ r3= x-mn
    addlt   r0, r3, r2      @   r0= x-mn + mx;
.Lend:
    ...
```

Cleans up nicely, wouldn't you say? Less branches, less code and it matches the C code more closely. We can even get rid of the last branch too because we can execute the `subs` conditionally as well. Because `ge` and `lt` are each others complements there won't be any interference. So the final version is:

```
@ Optimized wrapper, version 2
    subs    r3, r0, r2      @ r3= x-mx
    addge   r0, r3, r1      @   x= x-mx + mn
    sublts  r3, r0, r1      @ r3= x-mn
    addlt   r0, r3, r2      @   r0= x-mn + mx;
    ...
```

Of course, it isn't always possible to optimize to such an extent. However, if the clauses have small bodies conditional instructions may become attractive. Also, converting a compare to some form of data operation that you'd need later anyway is common and recommended.

Compound logical expressions

Higher languages often allow you to string multiple conditions together using logical AND (`&&`) and logical OR (`||`). What the books often won't say is that these are merely shorthand notations of a chain of `if`s. Here's what actually happens.

```
// === if(x && y) { /* clause */ } ===
if(x)
{
    if(y)
    { /* clause */ }
}

// === if(x || y) { /* clause */ } ===
if(x)
{ /* clause */ }
else if(y)
{ /* clause */ }
```

The later terms in the AND are only evaluated if earlier expressions were true. If not, they're simply skipped. Much fun can be had if the second term happens to be a function with side effects. A logical OR is basically an if-else chain with identical clauses; this is just for show of course, in the final version there's one clause which is branched to. In assembly, these would look something like this.

```
@ if(r0 != 0 && r1 != 0) { /* clause */ }
    cmp     r0, #0
    beq     .Lrest
    cmp     r1, #0
    beq     .Lrest
    @ clause
.Lrest:
    ...

@ Alternative
    cmp     r0, #0
    cmpne   r1, #0
    beq     .Lrest
    @ clause
.Lrest:
    ...
```

```
@ if( r0 != 0 || r1 != 0 ){ /* clause */ }
    cmp     r0, #0
    bne     .Ltrue
    cmp     r1, #0
    beq     .Lrest
.Ltrue:
    @ clause
.Lrest:
        ...
```

As always, alternative solutions will present themselves for your specific situation. Also note that you can transform ANDs into ORs using De Morgan's Laws.

Loops

One of the most important reasons to use assembly is speeding up oft-used code, which will probably involve loops because that's where most of the time will be spent. If you can remove one instruction in a non-loop situation, you'll have won one cycle. If you remove one from a loop, you'll have gained one for every iteration of the loop. For example, saving a cycle in a clear-screen function would save 240*160 = 19200 cycles – more, actually, because of memory wait-states. That one cycle can mean the difference between smooth and choppy animation.

In short, optimization is pretty much all about loops, especially inner loops. Interestingly, this is where GCC often misses the mark, because it adds more stuff than necessary. For example, in older versions (DKA and DKP r12, something like that) it often kept constructed memory addresses (VRAM, etc) inside the loop. Unfortunately, DKP r19 also has major issues with struct copies and `ldm`/`stm` pairs, which are now only give a small benefit over other methods.

Now, before you blame GCC for slow loops, it's also often the C programmer that forces GCC to produce slow code. In the introduction, you could see the major difference that inlining makes, and in the profiling demo I showed how much difference using the wrong datatype makes.

Anyway, loops in assembly. Making a loop is the easiest thing in the world: just branch to a previous label. The differences between `for`, `do-while` and `while` loops are a matter of where you increment and test. In C, you usually use a for-loop with an incrementing index. In assembly, it's customary to use a while-loop with a decrementing index. Here are two examples of a word-copy loop that should show you why.

```
@ Asm equivalents of copying 16 words.
@ u32 *dst=..., *src= ..., ii    // r0, r1, r2

@ --- Incrementing for-loop ---
@ for(ii=0; ii<16; ii++)
@     dst[ii]= src[ii];
    mov     r2, #0
.LabelF:
    ldr     r3, [r1, r2, lsl #2]
    str     r3, [r0, r2, lsl #2]
    add     r2, r2, #1
    cmp     r2, r2, #16
    blt .LabelF

@ --- Decrementing while-loop ---
@ ii= 16;
@ while(ii--)
@     *dst++ = *src++;
    mov     r2, #16
.LabelW:
    ldr     r3, [r1], #4
    str     r3, [r0], #4
    subs    r2, r2, #1
    bne .LabelW
```

In an incrementing for-loop you need to increment and then compare against the limit. In the decrementing while loop you subtract and test for zero. Because the zero-test is already part of every instruction, you don't need to compare separately. True, it's not much faster, maybe 10% or so, but many 10 percents here and there do add up. There are actually many versions of this kind of loop, here's another one using block-transfers. The benefit of those is that they also work in THUMB:

```
@ Yet another version, using ldm/stm

    add     r2, r0, #16
.LabelW:
    ldmia   r1!, {r3}
    stmia   r0!, {r3}
    cmp     r2, r0
    bne .LabelW
```

This is one of those occasions where knowing assembly can help you write efficient C. Using a decrementing counter and pointer arithmetic will usually be a little faster, but usually GCC willl do this for you anyway. Another point is using the right datatype. And with ‘right’ I mean `int`, of course. Non-ints require implicit casts (`lsl`/`lsr` pairs) after every arithmetic operation. That's two instructions after **every** plus, minus or whatever. While GCC has become quite proficient in converting non-ints into ints where possible, this has not always been the case, and it may not always be possible. I've seen the loops above cost between 600% more because the index and pointer were `u16`, I shit you not.

When dealing with loops, be extremely careful with how you start and stop the loop. It's very easy to come up with a loop that runs once too often or too little. I'm pretty sure these two versions are correct. The way I usually check it is to see how it runs when the count should be 1 or 2. If that works out, larger numbers will too.

Merge comparisons with data instructions

The **Z** status flag revolves around the number zero, so if you use 0 to compare to you can often combine the comparison with the data instruction that sets the flags.

This is also true for testing individual bits. The **N** and **C** flags are effectively bits 31 and 32, respectively. If you can manipulate the algorithm to use those, you don't need a `cmp` or `tst`.

“No, there is another”

You probably know this already, but this is a good time to repeat the warning: watch out for off-by-one errors (also known as obi-wan errors). It is just way too easy to do one iteration too few or too many, so always check whether the code you have does the right thing. Goes for other programming languages too, of course.

Function calls

Function calls use a special kind of branching instruction, namely `bl`. It works exactly like the normal branch, except that it saves the address after the `bl` in the link register (`r14` or `lr`) so that you know where to return to after the called function is finished. In principle, you can return with to the function using ‘`mov pc, lr`’, which points the program counter back to the calling function, but in practice you might be better off with `bx` (Branch and eXchange). The difference is that `bx` can also switch between ARM and THUMB states, which isn't possible with the `mov` return. Unlike `b` and `bl`, `bx` takes a register as its argument, instead of a label. This register will usually be `lr`, but the others are allowed as well.

There's also the matter of passing parameters to the function and returning values from it. In principle you're free to use any system you like, it is recommended to ARM's own ARM Architecture Procedure Call Standard (AAPCS) for this. For the majority of the work this can be summarized like this.

- The first 4 arguments go into r0-r3. Later ones go on the stack, in order of appearance.
- The return value goes into r0.
- The scratch registers r0-r3 (and r12) are free to use without restriction in a function. As such, after *calling* a function they should be considered ‘dirty’.
- The other registers must leave a function with the same values as they came in. Push them on the stack before use, and pop them when leaving the function. Note that another `bl` sets `lr`, so stack that one too in that case.

Below is a real-world example of function calling, complete with parameter passing, stackwork and returning from the call. The function `oamcpy()` copies OBJ_ATTRs. The function uses the same argument order as `memcpy()`, and these need to be set by the calling function; before and after the call, `lr` is pushed and popped. These two things are part of what's called the function overhead, which can be disastrous for small functions, as we've already seen. Inside `oamcpy()` we either jump back immediately if the count was 0, or proceed with the copies and then return. Note that `r4` is stacked here, because that's what the caller expects; if I hadn't and the caller used `r4` as well, I'd be screwed and rightly so. I should probably point out that `r12` is usually considered a scratch register as well, which I could have used here instead of `r4`, removing the need for stacking.

```
@ Function calling example: oamcpy
@ void oamcpy(OBJ_ATTR *dst, const OBJ_ATTR *src, u32 nn);
@ Parameters: r0= dst; r1= src; r2= nn;
    .align  2
oamcpy:
    cmp     r2, #0
    bxeq    lr          @ Nothing to do: return early
    push    {r4}        @ Put r4 on stack
.Lcpyloop:
        ldmia   r1!, {r3, r4}
        stmia   r0!, {r3, r4}
        subs    r2, #1
        bne     .Lcpyloop
    pop     {r4}        @ Restore r4 to its original value
    bx      lr          @ Return to calling function

@ Using oamcpy.
    @ Set arguments
    mov     r0, #0x07000000
    ldr     r1,=obj_buffer
    mov     r2, #128
    push    {lr}        @ Save lr
    bl      oamcpy      @ Call oamcpy (clobbers lr; assumes clobbering of r0-r3,r12)
    pop     {lr}        @ Restore lr
```

Use `bx` instead of `mov pc, lr`

The `bx` instruction is what makes interworking between ARM and THUMB function possible. Interworking is good. Therefore, `bx` is good.

This concludes the primary section on ARM assembly. There are more things like different processor states, and data-swap (`swp`) and co-processor instructions, but those are rare. If you need more information, look them up in the proper reference guides. The next two subsections cover instruction speeds and what an instruction actually looks like in binary, i.e., what the processor actually processes. Neither section is necessary in the strictest sense, but still informative. If you do not want to be informed, move on to the next section: the THUMB instruction set.

Cycle counting

Since the whole reason for coding in asm is speed (well, that and space efficiency), it is important to know how fast each instruction is so that you can decide on which one to use where. The term ‘cycle’ actually has two different meanings: there is the ***clock cycle***, which measures the amount of clock ticks, and there's ***functional cycle*** (for lack of a better word), which indicates the number of stages in an instruction. In an ideal world these two would be equal. However, this is the real world, where we have to deal with waitstates and buswidths, which make functional cycles cost multiple clock cycles. A ***wait(state)*** is the added cost for accessing memory; memory might just not be as fast as the CPU itself. Memory also as a fixed ***buswidths***, indicating the maximum number of bits that can be sent in one cycle: if the data you want to transfer is larger than the memory bus can handle, it has to be cut up into smaller sections and put through, costing additional cycles. For example, ROM has a 16bit bus which is fine for transferring bytes or halfwords, but words are transferred as two halfwords, costing two functional cycles instead of just one. If you hadn't guessed already, this is why THUMB code is recommended for ROM/EWRAM code.

There are three types of functional cycles: the ***non-sequential*** (N), the ***sequential*** (S) and the ***internal*** (I) cycle. There is a fourth, the coprocessor cycle (C), but as the GBA doesn't have a coprocessor I'm leaving that one out.

Anyway, the N- and S-cycles have to do with memory fetches: if the transfer of the current (functional) cycle is not related to the previous cycle, it is non-sequential; otherwise, it's sequential. Most instructions will be sequential (from the instruction fetch), but branches and loads/stores can have non-sequentials as they have to look up another address. Both sequential and non-sequential cycles are affected by section waitstates. The internal cycles is one where the CPU is already doing something else so that even though it's clear what the next action should be, it'll just have to wait. I-cycles do not suffer from waitstates.

Instruction	Cycles
Data	1S
ldr(type)	$1N + 1N_d + 1I$
str(type)	$1N + 1N_d$
ldm {*n*}	$1N + 1N_d + (n\text{-}1)S_d + 1I$
stm {*n*}	$1N + 1N_d + (n\text{-}1)S_d$
b/bl/bx/swi	2S + 1N
THUMB bl	3S + 1N
mul	1S + *m*I
mla/mull	1S + (*m*+1)I
mlal	1S + (*m*+2)I

Table 23.7: Cycle times for the most important instructions.

Section	Bus	Wait (N/S)	Access 8/16/32
BIOS	32	0/0	1/1/1
EWRAM	16	2/2	3/3/6
IWRAM	32	0/0	1/1/1
IO	32	0/0	1/1/1
PAL	16	0/0	1/1/2
VRAM	16	0/0	1/1/2
OAM	32	0/0	1/1/1
ROM	16	4/2	5/5/8

Table 23.8: Section default timing details. See also GBATek memory map.

Table 23.7 shows how much the instructions cost in terms of N/S/I cycles. How one can arrive to these cycle times is explained below. Table 23.8 lists the buswidths, the waitstates and the access times in clock cycles for each section. Note that these are the default wait states, which can be altered in REG_WAITCNT.

The data presented here is just an overview of the most important items, for all the gory details you should look them up in GBATek or the official documents.

- The cost of an instruction begins with fetching it from memory, which is a 1S operation. For most instructions, it ends there as well.

- Memory instructions also have to get data from memory, which will cost $1N_d$; I've added a subscript *d* here because this is an access to the section where the *data* is kept, whereas other waitstates are taken from the section where the code resides. This is an important distinction. Also, because the address of the next instruction won't be related to the current address, its timing will begin as a 1N instead of a 1S. This difference is encompassed in the transfer timing. Note however that most documentation list `ldr` as 1S+1N+1I, but this is false! If you actually test it, you'll see that it is really $1N+1N_d+1I$.
- Block transfer behave like normal transfers, except that all accesses after the first are S_d-cycles.
- Branches need an extra 1N+1S for jumping to the new address and fetching the resetting the pipeline (I think). Anything that changes `pc` can be considered a branch. The THUMB `bl` is actually two instructions (or, rather, one instruction and an address), which is why that has an additional 1S.
- Register-shifted operations add 1I to the base cost because the value has to be read from the register before it can be applied.
- Multiplies are 1S operations, plus 1I for every significant byte of the *second* operand. Yes, this does mean that the cost is asymmetric in the operands. If you can estimate the ranges of values for the operands, try to put the lower one in the second operand. Another 1I is required for the add of `mla`, and one more for long multiplications.

There is no 1S in loads!

Official documentation gives $1S+1N_d+1I$ as the timing of `ldr`, but this is not entirely accurate. It is actually 1**N**+$1N_d$+1I. The difference is small and only visible for ROM instructions, but could be annoying if you're wondering why what you predicted and what you measured for your routine doesn't match exactly. This applies to `ldm` and perhaps `swp` too.

See forum:9602 for a little more on the subject.

23.3.6. Anatomy of an addition

As an example of how instructions are actually formatted, I invite you to an in-depth look at the `add` instruction. This will be the absolute rock bottom in terms of GBA programming, the lowest level of them all. Understanding this will go a long way in understanding the hows and whys (and why-nots!) of ARM assembly.

Before I show the bits, I should point out that `add` (and in fact all data instructions) come in three forms, depending on the second operand. This can be an immediate field (numeric value), an immediate-shifted register or a register-shifted register. Bits 4 and 19h indicate the type of `add`, the lower 12 bits describe the second operand; the rest are the same for all `add` forms.

Table 23.3.6.: The `add` instruction(s)

	1F - 1C	1B 1A	19	18 - 15	14	13 - 10	F - C	B - 8	7	6 5	4	3 - 0
add Rd, Rn, #	**cnd**	**TA**	**IF**	**TB**	**S**	**Rn**	**Rd**	**IR**	**IN**			
add Rd, Rn, Rm Op #								**IS**		**ST**	**SF**	**Rm**
add Rd, Rn, Rm Op Rs								**Rs**	**0**			

Top 20 bits for `add`; denote instruction type, status/conditional flags and destination and first operand registers.

bits	name	description
C-F	Rd	**Destination register**.
10-13	Rn	**First operand register**.
14	S	Set **Status** bits (the `-s` affix).
15-18	TB	**Instruction-type** field. Must be 4 for `add` .
19	IF	**Immediate flag**. The second operand is an immediate value if set, and a (shifted) register if clear.
1A-1C	TA	Another **instruction-type** field. Is zero for all data instructions.
1D-1F	cnd	**Condition** field.

Lower 12 bits for `add`; these form the second operand.

bits	name	description
0-7	IN	**Immediate Number** field. The second operand is `IN ror 2*IR`.

8-B	IR	**Immediate Rotate** field. This denotes the rotate-right amount applied to *IN*.
0-3	Rm	**Second operand** register.
4	SF	**Shift-operand flag**. If set, the shift is the immediate value in *IS*; if clear, the shift comes from register *Rs*.
5-6	ST	**Shift type**. **0**: lsl, **1**: lsr **2**: asr, **3**: ror
7-B	IS	**Immediate Shift** value. Second operand is Rm Op IS.
8-B	Rs	**Shift Register**. Second operand is Rm Op Rs.

These kinds of tables should feel familiar: yes, I've also used them for IO-registers throughout Tonc. The fact of the matter is that the instructions are coded in a very similar way. In this case, you have a 32bit value, with different bitfields describing the type of instruction (TA=0 and TB=4 indicate is an add instruction), the registers to use (Rd, Rd and maybe Rm and Rs too) and a few others. We have seen this thing a number of times by now, so there should be no real difficulty in understanding here. The assembly instructions are little more than the BUILD macros I've used a couple of times, only this time it's the assembler that turn them into raw numbers instead of the preprocessor. Having said that, it *is* possible to construct the instructions manually, even at run-time, but whether you really want to do such a thing is another matter.

Now, the top 20 bits indicate the kind of instruction it is and which registers it uses. The bottom 12 are for *Op2*. If this involves a shifted register the bottom 4 bits indicate Rm. Bits 5 and 6 describe the type of shift-operation (shift-left, shift-right or rotate-right) and depending on bit 4, bits 7 to 11 form either the register to shift by (Rs) or a shift-value (5 bits for 0 to 31). And then there's the immediate operand …

Sigh. Yes, here are the mere twelve bits you can use for an immediate operand, divided into a 4bit rotate part and 8bit immediate part. The allowable immediate values are given by IN ror 2*IR. This seems like a small range, but interestingly enough you can get quite far with just these. It does mean that you can never load variable addresses into a register in one go; you have to get the address first with a PC-relative load and then load its value.

```
@ Forming 511(0x1FF)
    mov     r0, #511    @ Illegal instruction! D:

    mov     r0, #256    @ 256= 1 ror 24, so still valid
    add     r0, #255    @ 256+255 = 511

    @ Load 511 from memory with ldr
    ldr     r0, .L0

    @ Load 511 from memory with special ldr
    @ NOTE: no '#' !
    ldr     r0,=511
.L0:
    .word   511
```

Anyway, the bit patterns of table 23.9 is what the processor actually sees when you use an add instruction. You can see what the other instructions look like in the references I gave earlier, especially the quick references. The orthogonality of the whole instruction set shows itself in very similar formatting of a given class of instructions. For example, the data instructions only differ by the TB field: 4 for add, 2 for sub, et cetera.

23.4. THUMB assembly

The THUMB instruction set is a subset of the full list of ARM instructions. The defining feature of THUMB instructions is that they're only 16 bits long. As a result a function in THUMB can be much shorter than in ARM, which can be beneficial if you don't have a lot of room to work with. Another point is that 16bit instructions will pass through a 16bit databus in one go and be executed immediately, whereas execution of 32bit instructions would have to wait for the second chunk to be fetched, effectively halving the instruction speed. Remember that ROM and EWEAM, the two main areas for code have 16bit buses, which is why THUMB instructions are advised for GBA programming.

There are downsides, of course; you can't just cut the size of an instruction in half and expect to get away with it. Even though THUMB code uses many of the same mnemonics as ARM, functionality has been severely reduced. For example, the only instruction that can be conditional is the branch, b; instructions can no longer make use of shifts and rotates (these are separate instructions now), and most instructions can only use the lower 8 registers (r0-r7); the higher ones are still available, but you have to move things to the lower ones because you can use them.

In short, writing efficient THUMB code is much more challenging. It's not exactly bondage-and-disciple programming, but if you're used to the full ARM set you might be in for a surprise now and then. THUMB uses most of ARM's mnemonics, but a lot of them are restricted in some way so learning how to code in THUMB basically comes down to what you *can't* do anymore. With that in mind, this section will cover the differences between ARM and THUMB, rather than the THUMB set itself.

- **Removed instructions**. A few instructions have been cut altogether. Of the various multiplication instructions only `mul` remains, reverse subtractions (`rsb`, `rsc`) are gone, as are the swapping and coprocessor instructions, but those are rare anyway.
- **'New' instructions**. The mnemonics are new, but really, they're just special cases of conventional ARM instructions. THUMB has separate shift/rotate opcodes: `lsl`, `lsr`, `asr` and `ror` codes, which are functionally equivalent to '`mov Rd, Rm, Op2`'. There is also a '`neg Rd, Rm`' for *Rd*= 0−*Rm*, essentially an `rsb`. And I suppose you could call `push` and `pop` new, as they don't appear as ARM opcodes in some devkits.
- **No conditionals**. Except on branch. Hello, gratuitous labelling :\.
- The **Set Status** flag is always on. So in THUMB `sub` will always work as a `subs`, etc.
- **No barrel shifter**. Well, it still exist, of course; you just can't use it in conjunction with the instructions anymore. This is why there are separate bitshift/-rotate opcodes.
- **Restricted register availability**. Unless explicitly stated otherwise, the instructions can only use `r0-r7`. The exceptions here are `add`, `mov` and `cmp`, which can at times use high-regs as operands. This restriction also applies to memory operations, with small exceptions: `ldr/str` can still use PC-or SP-relative stuff; `push` allows `lr` in its register list and `pop` allows `pc`. With these, you could return from functions quickly, but you should use `bx` for that anyway. Fortunately, `bx` also allows use of every register so you can still do '`bx lr`'.
- **Little to no immediate or second operand support**. In ARM-code, most instructions allowed for a second operand *Op2*, which was either an immediate value or a (shifted) register. Most THUMB data instructions are of the form '*`ins`* `Rd, Rm`' and correspond to the C assignment operators like `+=` and `|=`. Note that `Rm` is a register, not an immediate. The only instructions that break this pattern are the shift-codes, `add`, `sub`, `mov` and `cmp`, which can have both immediate values and second operands. See the reference docs for more details.
- **No write-back in memory instructions**. That means you will have to use at least one extra register and extra instructions when traversing arrays. There is one exception to this, namely block-transfers. The only surviving versions are `ldmia` and `stmia`, and in both versions the write-back is actually required.
- **Memory operations are tricky**. Well, they are! ARM memory opcodes were identical in what they could do, but here you have to be on your toes. Some features are completely gone (write-back and shifted register offsets), and the others aren't always available to all types. Register-offset addressing is always available, but immediate offsets do not work for the signed loads (`ldrsh`, `ldrsb`). Remember that the registers can only be `r0-r7`, except for `ldr/str`: there you can also use PC and SP-relative stuff (with immediate offsets). Table 23.10 gives an overview.

	[Rn,Rm]	[Rn,#]	[pc/sp,#]
ldr/str	+	+	+
ldrh/strh	+	+	-
ldrb/strb	+	+	-
ldrsh/ldrsb	+	-	-

Table 23.10. THUMB addressing mode availability.

Actually, '`ldrh Rd, =X`' also seem to work, but these are actually converted into '`ldr Rd, =X`' internally.

Is that it? Well no, but it's enough. Remember, THUMB is essentially ARM Lite: it looks similar, but it has lost a lot of substance. I'd suggest learning THUMB code in that way too: start with ARM then learn what you can't do anymore. This list gives most of the things you need to know; for the rest, just read at the assembler messages you will get from time to time and learn from the experience.

23.5. GAS: the GNU assembler

The instructions are only part of functional assembly, you also need ***directives*** to tie code together, control sections and alignment, create data, etc. Somewhat fittingly, directives seem to be as unportable as assembly itself: directives for one assembler might not work under others.

This section covers the main directives of the GNU assembler, GAS. Since we're already working with the GNU toolchain, the choice for this assembler is rather obvious. GAS is already part of the normal build sequence, so there is no real loss of functionality, and you can work together with C files just as easily as with other assembly; it's all the same to GCC. Another nice feature is that you can use the preprocessor so if you have header files with just preprocessor stuff

(#include and #define only), you can use those here as well. Of course, you could do that anyway because `cpp` is a standalone tool, but you don't have to resort to trickery here.

But back to directives. In this section you'll see some of the most basic directives. This includes creating symbols for functions (both ARM and THUMB) and variables. With out these you wouldn't be able to do anything. I'll also cover basic datatypes and placing things in specific sections. There are many other directives as well, but these should be the most useful. For the full list, go to the GAS manual at www.gnu.org.

23.5.1. Symbols

Data (variable and constant) and functions are collectively known as ***symbols*** and, just like in C, these have declarations and definitions. Any label (a valid identifier on a separate line ending with a colon) is potentially a symbol definition, but it's better to distinguish between global and local labels. Simply put, a label is global if there is a ‘`.global` *lname*’ directive attached to it that makes it visible to the outside world. Local labels are everything else, and conventionally start with ‘`.L`’, though this is not required. If you want to use symbols from outside, you have to use ‘`.extern` *lname*’.

Now, unless you're using some notational device, a label tells you nothing about what it actually stands for: it gives you no information on whether it's data or a function, nor the number of arguments for the latter. There is ‘`.type,` *str*’ directive that lets you indicate if it's a function (*str* = `%function`) or some form of data (*str* = `%object`), but that's about it. Since you can tell that difference by looking at what's after the label anyway, I tend to leave this out. For other information, please comment on what the symbols mean.

The directives you'd use for data will generally tell you what the datatypes are, but that's something for a later subsection. Right now, I'll discuss a few directives for functions. The most important one is ‘`.code` *n*’, where *n* is 32 or 16 for ARM or THUMB code respectively. You can also use the more descriptive `.arm` and `thumb` directives, which do the same thing. These are global settings, remaining active until you change them. Another important directive is `.thumb_func`, which is **required** for interworking THUMB functions. This directive applies to the next symbol label. Actually, `.thumb_func` already implies `.thumb`, so adding the latter explicitly isn't necessary.

A very important and sneaky issue is alignment. **You** are responsible for aligning code and data, not the assembler. In C, the compiler did this for you and the only times you might have had problems was with alignment mismatches when casting, but here both code *and* data can be misaligned; in assembly, the assembler just strings your code and data together as it finds it, so as soon as you start using anything other than words you have the possibility of mis-alignments.

Fortunately, alignment is very easy to do: ‘`.align` *n*’ aligns to the next 2^n byte boundary and if you don't like the fact that *n* is a power here, you can also use ‘`.balign` *m*’, which aligns to *m* bytes. These will update the current location so that the next item of business is properly aligned. Yes, it applies to the *next* item of code/data; it is not a global setting, so if you intend to have mixed data-sizes, be prepared to align things often.

Here are a few examples of how these things would work in practice. Consider it standard boilerplate material for the creation and use of symbols.

```
@ ARM and THUMB versions of m5_plot
@ extern u16 *vid_page;
@ void m5_plot(int x, int y, u16 clr)
@ {   vid_page[y*160+x]= clr;    }

@ External declaration
@ NOTE: no info on what it's a declaration of!
    .extern vid_page            @ extern u16 *vid_page;

@ ARM function definition
@ void m5_plot_arm(int x, int y, u16 clr)
    .align 2                    @ Align to word boundary
    .arm                        @ This is ARM code
    .global m5_plot_arm         @ This makes it a real symbol
    .type m5_plot_arm STT_FUNC  @ Declare m5_plot_arm to be a function.
m5_plot_arm:                    @ Start of function definition
    add     r1, r1, lsl #2
    add     r0, r1, lsl #5
    ldr     r1,=vid_page
    ldr     r1, [r1]
    mov     r0, r0, lsl #1
    strh    r2, [r1, r0]
    bx      lr

@ THUMB function definition
@ void m5_plot_thumb(int x, int y, u16 clr)
    .align 2                    @ Align to word boundary
    .thumb_func                 @ This is a thumb function
    .global m5_plot_thumb       @ This makes it a real symbol
    .type m5_plot_thumb STT_FUNC    @ Declare m5_plot_thumb to be a function.
m5_plot_thumb:                  @ Start of function definition
    lsl     r3, r1, #2
    add     r1, r3
    lsl     r1, #5
    add     r0, r1
    ldr     r1,=vid_page
    ldr     r1, [r1]
    lsl     r0, #1
    strh    r2, [r1, r0]
    bx      lr
```

The functions above show the basic template for functions: three lines of directives, and a label for the function. Note that there is no required order for the four directives, so you may see others as well. In fact, the `.global` directive can be separated completely from the rest of the function's code if you want. Also note the use of `.extern` to allow access to `vid_page`, which in tonclib always points to the current back buffer. To be honest, it isn't even necessary because GAS assumes that all unknown identifiers come from other files; nevertheless, I'd suggest you use it anyway, just for maintenance sake.

And yes, these two functions do actually form functional mode 5 pixel plotters. As an exercise, try to figure out how they work and why they're coded the way they are. Also, notice that the THUMB function is only two instructions longer than the ARM version; if this were ROM-code, the THUMB version would be a whole lot faster due to the buswidth, which is exactly why THUMB code is recommended there.

GCC 4.7 note: symbol-type for functions now required

As of GCC 4.7, the `.type` directive is pretty much required for functions. Or, rather, it is required if you want ARM and Thumb interworking to work. Just add the following line to each function definition:

```
    .type [function-name] STT_FUNC
```

`STT_FUNC` is an internal macro that expands to the correct attribute (presumably `%function`). Replace `[function-name]` with the real function name.

Implicit extern considered harmful

The `.extern` directive for external symbols is actually not required: GAS assumes that unknown identifiers are external. While I can see the benefits of implicit declarations/definitions, I still think that it is a *bad* idea. If you've ever misspelled an identifier in languages that have implicit definitions, you'll know why.

And yes, I know this is actually a non-issue because it'll get caught by the linker anyway, but explicitly mentioning externals is probably still a good idea. `:P`

23.5.2. Definition of variables

Of course, you can also have data in assembly, but before we go there a word about acceptable number formats and number uses. GAS uses the same number representations as C: plain numbers for decimals, ‘`0`’ for octal and ‘`0x`’ for hexadecimal. There is also a binary representation, using the ‘`0b`’ prefix: for example, `0b1100` is 12. I've already used numbers a couple of times now, and you should have noticed that they're sometimes prepended by ‘`#`’. The symbol is not actually part of the number, but is the indicator for an immediate value.

It is also possible to perform arithmetic on numbers in assembly. That is to say, you can have something like ‘`mov r0, #1+2+3+4`’ to load 10 into `r0`. Not only arithmetic, but bit operations will work too, which can be handy if you want to construct masks or colors. Note, this only works for constants.

And now for adding data to your code. The main data directives are `.byte`, `.hword`, and `.word`, which create bytes, halfwords and words, respectively. If you want you can count `.float` among them as well, but you don't want to because floats are evil on the GBA. Their use is simple: put the directive on a line and add a number after it, or even a comma-separated list for an array. If you add a label in front of the data, you have yourself a variable. There are also a few directives for strings, namely `.ascii`, `.asciz` and `.string`. `.asciz` and `.string` differ from `.ascii` in that they add the terminating '`\0`' to the string, which is how strings usually work. Just like the other data directives, you can make an array of strings by separating them with commas.

You can see some examples below; note that what should have been the hword_var will definitely be misaligned and hence useless.

```
    .align 2
word_var:               @ int word_var= 0xCAFEBABE
    .word   0xCAFEBABE
word_array:             @ int word_array[4]= { 1,2,3,4 }
    .word   1, 2, 3, 4      @ NO comma at the end!!
byte_var:               @ char byte_var= 0;
    .byte   0
hword_var:              @ NOT short hword_var= 0xDEAD;
    .hword  0xDEAD      @   due to bad alignment!
str_array:                 @ Array of NULL-terminated strings:
    .string "Hello", "Nurse!"
```

23.5.3. Data sections

So now you know how to make code and variables, you have to put them into the proper sections. A ***section*** is a contained area where code and data are stored; the linker uses a linkscript to see where the different sections are and then adds all your symbols and links accordingly. The format for sections is ‘.section *secname*’, with optional ‘, *"flags"*, *%type*’ information I'll get to in a minute.

Traditionally, the section for code is called .text and that for data is called .data, but there are a few more to consider: the general sections .bss and .rodata, and the GBA-specific .ewram and .iwram. In principle, these four are data sections, but they can be used for code by setting the correct section flags. As you might have guessed, .ewram stands for the EWRAM section (0200:0000h), .iwram is IWRAM (0300:0000h) and .rodata is ROM (0800:0000). The .bss section is a section intended for variables that are either uninitialized or filled with zero. The nice thing about this section is that it requires no ROM space, as it doesn't have data to store there. The section will be placed in IWRAM, just like the .data. You may also sometimes see .sbss which stands for ‘small bss’ and has a similar function as the standard .bss, but happens to be placed in EWRAM.

These data sections can be used to indicate different kinds of data symbols. For example, constants (C keyword const) should go into .rodata. Non-zero (and non-const, obviously) initialised data goes into .data, and zero or uninitialized data is to be placed into .bss. Now, you still have to indicate the amount of storage you need for each bss-variable. This can be done with ‘.space *n*’, which indicates *n* zero bytes (see also .fill and .skip), or ‘.comm *name*, *n*, *m*’, which creates a bss symbol called *name*, allocates *n* bytes for it and aligns it to *m* bytes too. GCC likes to use this for uninitialized variables.

```
// C symbols and their asm equivalents

// === C versions ===
int var_data= 12345678;
int var_zeroinit= 0;
int var_uninit;
const u32 cst_array[4]= { 1, 2, 3, 4 };
u8 charlut[256] EWRAM_BSS;

@ === Assembly versions ===
@ Removed alignment and global directives for clarity

@ --- Non-zero Initialized data ---
    .data
var_data:
    .word   12345678

@ -- Zero initialized data ---
    .bss
var_zeroinit:
    .space      4

@ --- Uninitialized data ---
@ NOTE: .comm takes care of section, label and alignment for you
@   so those things need not be explicitly mentioned
    .comm var_uninit,4,4

@ --- Constant (initialized) data ---
    .section .rodata
```

```
cst_array:
    .word 1, 2, 3, 4

@ --- Non-zero initialized data in ewram ---
    .section .sbss
charlut:
    .space 256
```

Assembly for data exporters

Assembly is a good format for exporting data to. Assembling arrays is faster than compilation, the files can be bigger and you can control alignment more easily. Oh, any you can't be tempted to #include the data, because that simply will not work.

```
    .section .rodata    @ in ROM, please
    .align  2           @ Word alignment
    .global foo         @ Symbol name
foo:
    @ Array goes in here. Type can be .byte, .hword or .word
    @ NOTE! No comma at the end of a line! This is important
    .hword  0x0000,0x0001,0x0002,0x0003,0x0004,0x0005,0x0006,0x0007
    .hword  0x0008,0x0009,0x000A,0x000B,0x000C,0x000D,0x000E,0x000F
    .hword  0x0010,0x0011,0x0012,0x0013,0x0014,0x0015,0x0016,0x0017

    ...
```

You need a const section, word alignment, a symbol declaration and definition and the data in soe form of array. To make use of it, make a suitable declaration of the array in C and you're all set.

23.5.4. Code sections

That was data in different sections, now for code. The normal section for code is `.text`, which will equate to ROM or EWRAM depending on the linker specs. At times, you will want to have code in IWRAM because it's a lot faster than ROM and EWRAM. You might think that ‘`.section .iwram` does the trick, but unfortunately this does not seem generally true. Because IWRAM is actually a data section, you have to add section-type information here as well. The full section declaration needs to be ‘`.section .iwram, "ax", %progbits`’, which marks the section as allocatable and executable (`"ax"`), and that the section contains data as well (`%progbits`), although this last bit doesn't seem to be required.

Another interesting point is how to call the function once you have it in IWRAM. The problem is that IWRAM is too far away from ROM to jump to in one go, so to make it work you have to load the address of your function in a register and then jump to it using `bx`. And set `lr` to the correct return address, of course. The usual practice is to load the function's address into a register and branch to a dummy function that just consists of a `bx` using that register. GCC has these support functions under the name `_call_via_`*`rx`*, where *rx* is the register you want to use. These names follow the GCC naming scheme as given in table 23.1.

```
@ --- ARM function in IWRAM: ---
    .section .iwram, "ax", %progbits
    .align 2
    .arm
    .global iw_fun
    .type iw_fun STT_FUNC
iw_fun:
    @ <code goes in here>


@ --- Calling iw_fun somewhere ---
    ldr r3,=iw_fun      @ Load address
    bl  _call_via_r3   @ Set lr, jump to long-call function


@ --- Provided by GCC: ---
_call_via_r3:
    bx  r3        @ Branch to r3's address (i.e., iw_fun)
                  @ No bl means the original lr is still valid
```

The `_call_by_rx` indirect branching is how function calling works when you use the `-mlong-calls` compiler flag. It's a little slower than straight branching, but generally safer. Incidentally, this is also how interworking is implemented.

Standard and special sections

Sections `.text`, `.data` and `.bss` are standard GAS sections and do not need explicit mention of the `.section` directive. Sections `.ewram`, `.iwram`, `.rodata` and `.sbss` and more GBA specific, and do need `.section` in front of them.

With this information, you should be able to create functions, variables and even place them into the right sections. This is probably 90% of whay you might want to do with directives already. For the remaining few, read the manual or browse through GCC generated asm. Both should point you in the right direction.

23.6. A real world example: fast 16/32-bit copiers

In the last section, I will present two assembly functions – one ARM and one THUMB – intended for copying data quickly and with safety checks for alignment. They are called `memcpy16()` and `memcpy32()` and I have already used these a number of times throughout Tonc. `memcpy32()` does what `CpuFastSet()` does, but without requiring that the word-count is a multiple of 8. Because this is a primary function, it's put in IWRAM as ARM code. `memcpy16()` is intended for use with 16bit data and calls `memcpy32()` if alignments of the source and destination allow it and the number of copies warrant it. Because its main job is deciding whether to use `memcpy32`, this function can stay in ROM as THUMB code.

This is not merely an exercise; these functions are there to be used. They are optimized and take advantage of most of the features of ARM/THUMB assembly. Nearly everything covered in this chapter can be found here, so I hope you've managed to keep up. To make things a little easier, I've added the C equivalent code here too, so you can compare the two.

Also, these functions are not just for pure assembly projects, but can also be used in conjunction with C code, and I'll show you how to do this too. As you've already seen demos using these functions without any hint that they were assembly functions (apart from me saying so), this part isn't actually to hard. So that's the program for this section. Ready? Here we go.

23.6.1. memcpy32()

This function will copy words. *Fast*. The idea is to use 8-fold block-transfers where possible (just like `CpuFastSet()`), and then copy the remaining 0 to 7 words of the word count with simple transfers. Yes, one could make an elaborate structure of tests and block-transfers to do these residuals in one go as well, but I really don't think that's worth it.

One could write a function for this in C, which is done below. However, even though GCC does use block-transfers for the BLOCK struct-copies, I've only seen it go up to 4-fold `ldm/stm`s. Furthermore, it tends to use more registers than strictly necessary. You could say that GCC doesn't do its job properly, but it's hard to understand what humans mean, alright? If you want it done as efficient as possible, do it your damn self. Which is exactly what we're here to do, of course.

```c
// C equivalent of memcpy32
typedef struct BLOCK { u32 data[8]; } BLOCK;

void memcpy32(void *dst, const void *src, uint wdcount) IWRAM_CODE
{
    u32 blkN= wdcount/8, wdN= wdcount&7;
    u32 *dstw= (u32*)dst, *srcw= (u32*)src;
    if(blkN)
    {
        // 8-word copies
        BLOCK *dst2= (BLOCK*)dst, *src2= (BLOCK*)src;
        while(blkN--)
            *dst2++ = *src2++;
        dstw= (u32*)dst2;  srcw= (u32*)src2;
    }
    // Residual words
    while(wdN--)
        *dstw++ = *srcw++;
}
```

The C version should be easy enough to follow. The number of words, `wdcount` is split into a block count and residual word count. If there are full block to copy, then we do so and adjust the pointers so that the residuals copy correctly. Note that `wdcount` is the number of words to copy, not the number of bytes, and that `src` and `dst` are assumed to be word-aligned.

The assembly version –surprise, surprise– does exactly the same thing, only much more efficient than GCC will make it. There is little I can add about what it does because all things have been covered already, but there are a few things about *how* it does them that deserve a little more attention.

First, note the general program flow. The `movs` gives the number of blocks to copy, and if that's zero then we jump immediately to the residuals, `.Lres_cpy32`. What happens there also interesting: the three instructions after decrementing the word-count (in `r12`) all carry the `cs` flag. This means that if `r12` is (unsigned) lower than one, (i.e., `r12`==0) these instructions are ignored. This is exactly what we want, but usually there is a separate check for zero-ness before the body of the loop and these extra instructions cost space and time. With clever use of conditionals, we can spare those.

The main loop doesn't use these conditionals, nor, it would seem, a zero-check. The check here is actually done at that `movs` lines as well: if it doesn't jump, we can be sure there are blocks to copy, so another check is unnecessary. Also note that the non-scratch registers `r4-r10` are only stacked when we're sure they'll actually be used. GCC normally stacks at the beginning and end of functions, but there is no reason not to delay it until it's actually necessary.

Lastly, a few words on non-assembly matters. First, the general layout: I use one indent for everything except labels, and sometimes more for what in higher languages would be loops or if-blocks. Neither is required, but I find that it makes reading easier. I also make sure that the instruction parameters are all in line, which works best if you reserve 8 spaces for the mnemonic itself. How you set the indents is a matter of personal preference and a subject of many holy wars, so I'm not touching that one here `:P`

Another point is comments. Comments are even more important in assembly than in C, but don't overdo it! Overcommenting just drowns out the comments that are actually useful and maybe even the code as well. Comment on blocks and what each register is and maybe on important tests/branches, but do you really have to say that ‘`subs r2, r2, #1`’ decrements a loop variable? No, I didn't think so either. It might also help to give the intended declaration of the function if you want to use it in C.

Also, it's a good idea to always add section, alignment and code-set before a function-label. Yes, these things aren't strictly necessary, but what if some yutz decides to add a function in the middle of the file which screws up these things for functions that follow it? Lastly, try to distinguish between symbol-labels and branch-labels. GCC's take on this is starting the latter with ‘`.L`’, which is as good of a convention as any.

```
@ === void memcpy32(void *dst, const void *src, uint wdcount) IWRAM_CODE; =============
@ r0, r1: dst, src
@ r2: wdcount, then wdcount>>3
@ r3-r10: data buffer
@ r12: wdn&7
    .section .iwram,"ax", %progbits
    .align  2
    .code   32
    .global memcpy32
    .type   memcpy32 STT_FUNC
memcpy32:
    and     r12, r2, #7     @ r12= residual word count
    movs    r2, r2, lsr #3  @ r2=block count
    beq     .Lres_cpy32
    push    {r4-r10}
    @ Copy 32byte chunks with 8fold xxmia
    @ r2 in [1,inf>
.Lmain_cpy32:
        ldmia   r1!, {r3-r10}
        stmia   r0!, {r3-r10}
        subs    r2, #1
        bne     .Lmain_cpy32
    pop     {r4-r10}
    @ And the residual 0-7 words. r12 in [0,7]
.Lres_cpy32:
        subs    r12, #1
        ldrcs   r3, [r1], #4
        strcs   r3, [r0], #4
        bcs     .Lres_cpy32
    bx  lr
```

23.6.2. memcpy16()

The job of the halfword copier memcpy16() isn't really copying halfwords. If possible, it'll use memcpy32() for addresses that can use it, and do the remaining halfword parts (if any) itself. Because it doesn't do much copying on its own we don't have to waste IWRAM with it; the routine can stay as a normal THUMB function in ROM.

Two factors decide whether or not jumping to memcpy32() is beneficial. First is the number of halfwords (hwcount) to copy. I've ran a number of checks and it seems that the break-even point is about 6 halfwords. At that point, the power of word copies in IWRAM already beats out the cost of function-call overhead and THUMB/ROM code.

The second is whether the incoming source and destination addresses can be resolved to word addresses. This is true if bit 1 of the source and destinations are equal (bit 0 is zero because these are valid halfword addresses), in other words: (src^dst)&2 should not be zero. If it resolvable, do one halfword copy to word-align the addresses if necessary, then call memcpy32() for all the word copies. After than, adjust the original halfword stuff and if there is anything left (or if memcpy32() couldn't be used) copy by halfword.

```
// C equivalent of memcpy16
void memcpy16(void *dst, const void *src, uint hwcount)
{
    u16 *dsth= (u16*)dst, *srch= (u16*)src;
    // Fast-copy if and only if:
    //    (1) enough halfwords and
    //    (2) equal src/dst alignment
    if( (hwcount>5) && !(((u32)dst^(u32)src)&2) )
    {
        if( ((u32)src)&1 )  // (3) align to words
        {
            *dsth++= *srch++;
            hwcount--;
        }
        // (4) Use memcpy32 for main stint
        memcpy32(dsth, srch, hwcount/2);
        // (5) and adjust parameters to match
        srch += hwcount&~1;
        dsth += hwcount&~1;
        hwcount &= 1;
    }
    // (6) Residual halfwords
    while(hwcount--)
        *dsth++ = *srch++;
}
```

The C version isn't exactly pretty due to all the casting and masking, but it works well enough. If you were to compile it and compare it to the assembly below you should see many similarities, but it won't be exactly equal because the assembly programmer is allowed greater freedoms than GCC, and doesn't have to contend with the syntax of C.

Anyway, before the function actually starts I state the declaration, the use of the registers, and the standard boilerplate for functions. As I need more than 4 registers and I'm calling a function, I need to stack r4 and lr. This time I am doing this at the start and end of the function because it's just too much of a hassle not to. One thing that may seem strange is why I pop r4 and then r3 separately, especially as it's lr that I need and not r3. Remember the register restrictions: lr is actually r14, which can be reached by push, but not pop. So I'm using r3 here instead. I'm also doing this separately from the r4-pop because 'pop {r4, r3}' pops the registers in the wrong order (lower regs are loaded first).

The rest of the code follows the structure of the C code; I've added numbered points to indicate where we are. Point 1 checks the size, and point 2 checks the relative alignment of source and destination. Note that what I actually do here is not AND with 2, but shift by 31, which pushes bit 1 into the carry bit; THUMB code can only AND between registers and rather than putting 2 in a register and ANDing, I just check the carry bit. You can also use the sign bit to shift to, of course, which is what GCC would do. I do something similar to check whether the pointers are already word-aligned or if I have to do that myself.

At point 4 I set up and call memcpy32(). Or, rather, I call _call_via_r3, which calls memcpy32(). I can't use 'bl memcpy32' directly because its in IWRAM and memcpy16() is in ROM and the distance is simply too big. The _call_via_r3 is an intermediary (in ROM) consisting only of 'bx r3' and since memcpy32()'s address was in r3 we got where we wanted to go. Returning from memcpy32() will work fine, as that was set by the call to _call_via_r3.

The C code's point 5 consisted of adjusting the source and destination pointers to account for the work done by memcpy32(); in the assembly code, I'm being a very sneaky bastard by not doing any of that. The thing is, after memcpy32() is done r0 and r1 would *already* be where I want them; while the rules say that r0-r3 are clobbered by calling functions and should therefore be stacked, if I *know* that they'll only end up the way I want them, do I really have to do the extra work? I think not. Fair enough, it's not recommended procedure, but where's the fun in asm programming if you can't cheat a little once in a while? Anyway, the right-shift from r4 counters the left-shift into r4 that I had before, corresponding to a r2&1; the test after it checks whether the result is zero, signifying that I'm done and I can get out now.

Lastly, point 6 covers the halfword copying loop. I wouldn't have mentioned it here except for one little detail: the array is copied back to front! If this were ARM code I'd have used post-indexing, but this is THUMB code where no such critter exists and I'm restricted to using offsets. I could have used another register for an ascending offset (one extra instruction/loop), or incrementing the r0 and r1 (two extra per loop), or I could copy backwards which works just as well. Also note that I use bcs at the end of the loop and not bne; bcs is essential here because r2 could already be 0 on the first count, which bne would miss.

```
@ === void memcpy16(void *dst, const void *src, uint hwcount); =============
@ Reglist:
@  r0, r1: dst, src
@  r2, r4: hwcount
@  r3: tmp and data buffer
    .text
    .align  2
    .code   16
    .thumb_func
    .global memcpy16
    .type   memcpy16 STT_FUNC
memcpy16:
    push    {r4, lr}
    @ (1) under 5 hwords -> std cpy
    cmp     r2, #5
    bls     .Ltail_cpy16
    @ (2) Unreconcilable alignment -> std cpy
    @ if (dst^src)&2 -> alignment impossible
    mov     r3, r0
    eor     r3, r1
    lsl     r3, #31         @ (dst^src), bit 1 into carry
    bcs     .Ltail_cpy16    @ (dst^src)&2 : must copy by halfword
    @ (3) src and dst have same alignment -> word align
    lsl     r3, r0, #31
    bcc     .Lmain_cpy16    @ ~src&2 : already word aligned
    @ Aligning is necessary: copy 1 hword and align
        ldrh    r3, [r1]
        strh    r3, [r0]
        add     r0, #2
        add     r1, #2
        sub     r2, #1
    @ (4) Right, and for the REAL work, we're gonna use memcpy32
.Lmain_cpy16:
    lsl     r4, r2, #31
    lsr     r2, r2, #1
    ldr     r3,=memcpy32
    bl      _call_via_r3
    @ (5) NOTE: r0,r1 are altered by memcpy32, but in exactly the right
    @ way, so we can use them as is.
    lsr     r2, r4, #31
    beq     .Lend_cpy16
    @ (6) Copy residuals by halfword
.Ltail_cpy16:
    sub     r2, #1
    bcc     .Lend_cpy16     @ r2 was 0, bug out
    lsl     r2, r2, #1      @ r2 is offset (Yes, we're copying backward)
.Lres_cpy16:
        ldrh    r3, [r1, r2]
        strh    r3, [r0, r2]
        sub     r2, r2, #2
        bcs     .Lres_cpy16
.Lend_cpy16:
    pop     {r4}
    pop     {r3}
    bx  r3
```

Using memcpy32() and memcpy16() in C

While you're working on uncrossing your eyes, a little story on how you can call these functions from C. It's ridiculously simple actually: all you need is a declaration. Yup, that's it. GCC does really care about the language the functions are in, all it asks is that they have a consistent memory interface, as covered in the AAPCS. As I've kept myself to this standard (well, mostly), there is no problem here.

```
// Declarations of memcpy32() and memcpy16()
void memcpy16(void *dst, const void *src, uint hwcount);
void memcpy32(void *dst, const void *src, uint wdcount) IWRAM_CODE;

// Example use
{
    extern const u16 fooPal[256];
    extern const u32 fooTiles[512];

    memcpy16(pal_bg_mem, fooPal, 256);      // Copy by halfword. Fine
    memcpy32(pal_bg_mem, fooPal, 256/2);    // Copy by word; Might be unsafe
    memcpy32(tile_mem, fooTiles, 512);      // Src is words too, no prob.
}
```

See? They can be called just as any other C function. Of course, you have to assemble and link the assembly files instead of #include them, but that's how you're supposed to build projects anyway.

You do need to take care to provide the **correct** declaration, though. The declaration tells the compiler how a function expects to be called, in this case destination and source pointers (in that order), and the number of (half)words to transfer. It is legal to change the order of the parameters or to add or remove some – the function won't *work* anymore, but the compiler does allow it. This is true of C functions too and should be an obvious point, but assembly functions don't provide an easy check to determine if you've used the correct declaration, so please be careful.

Make the declaration fit the function

This is a very important point. Every function has expectations on how it's called. The section, return-type and arguments of the declaration must match the function's, lest chaos ensues. The best way would be not explicitly mention the full declaration near the function definition, so that the user just has to copy-paste it.

Use `extern "C"' for C++

Declarations for C++ work a little different, due to the name mangling it expects. To indicate that the function name is *not* mangled, add ‘`extern ″C″`’ to the declaration.

```
// C++ declarations of memcpy32() and memcpy16()
extern "C" void memcpy16(void *dst, const void *src, uint hwcount);
extern "C" void memcpy32(void *dst, const void *src, uint wdcount) IWRAM_CODE;
```

And that all folks. As far as this chapter goes anyway. Like all languages, it takes time to fully learn its ins and outs, but with this information and a couple of (quick)reference documents, you should be able to produce some nice ARM assembly, or at least be able to read it well enough. Just keep your wits about you when writing assembly, please. Not just in trying to avoid bugs, but also in keeping the assembly maintainable. Not paying attention in C is bad enough, but here it can be absolutely disastrous. Think of what you want to do first, *then* start writing the instructions.

Also remember: yes, assembly can be fun. Think of it as one of those shuffle-puzzles: you have a handful of pieces (the registers) and ways of manipulating them. The goal is to get to the final picture in then least amount of moves. For example, take a look at what an optimized palette blend routine would look like. Now it's your turn `:P`.

Modified Jan 28, 2015, J Vijn. Get all Tonc files here