**An introduction to**

**Cortex M0 assembler programming**

**using GCC**



Wouter van Ooijen (wouter@voti.nl), 2019-08-27, 1.07

# Summary

This document introduces the Cortex M0 assembly language for the GNU assembler, the interaction between assembly code and C++ code, and the mechanism of context switching. The reader is assumed to have a basic understanding of computer architecture and the C and C++ languages.

Contents

[Summary 1](#_Toc523493200)

[1. Low-level computer languages 3](#_Toc523493201)

[1.1. Machine language 3](#_Toc523493202)

[1.2. Assembly language 3](#_Toc523493203)

[1.3. Data path size 4](#_Toc523493204)

[1.4. Instruction set design 4](#_Toc523493205)

[1.5. Memory use 6](#_Toc523493206)

[2. A bit of history 7](#_Toc523493207)

[2.1. Acorn computers to Arm Ltd. 7](#_Toc523493208)

[2.2. The low-power market 8](#_Toc523493209)

[2.3. From 32 bits ARM to 16 bits Cortex 9](#_Toc523493210)

[3. Cortex M0 programming model 9](#_Toc523493211)

[3.1. Registers 9](#_Toc523493212)

[3.2. Pipelining 10](#_Toc523493213)

[4. Cortex M0 assembly language 13](#_Toc523493214)

[4.1. Literal values 13](#_Toc523493215)

[4.2. Directives 13](#_Toc523493216)

[4.3. Load, store and register-to-register move instructions 14](#_Toc523493217)

[4.4. Calculations 16](#_Toc523493218)

[4.5. Flow control instructions 19](#_Toc523493219)

[4.6. Miscellaneous instructions 21](#_Toc523493220)

[5. Subroutines 22](#_Toc523493221)

[5.1. Calling Standard 22](#_Toc523493222)

[5.2. Local variables 24](#_Toc523493223)

[5.3. Combining C and assembly 25](#_Toc523493224)

[6. Startup 27](#_Toc523493225)

[7. Building an application 29](#_Toc523493226)

[8. Context switching 32](#_Toc523493227)

[8.1. Cooperative multithreading 32](#_Toc523493228)

[8.2. Preemptive multithreading 35](#_Toc523493229)

[9. Further reading 37](#_Toc523493230)

# Low-level computer languages

## Machine language

A computer is a device that performs a function that is fixed not in its construction, but in a form of data that is somehow stored in it. Modern computers can be schematically divided in three parts: the memory that stores data, processor or CPU that retrieves data from the memory and executes it, and the I/O devices that interface to the world outside the computer.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Memory  (ROM, RAM) |  | CPU (processor) |  | I/O  (interface to the real world) |
|  |  |  |  |  |
|  | | | | |

The instructions fetched by the CPU from memory to be subsequently executed are in a format that is convenient for the CPU, not for a human. It must be compact, because there is a limit to the speed at which the memory can supply data, so if the format were verbose this would limit the overall speed of the computer. It must also be easy for the CPU to interpret; otherwise the interpretation would be a bottleneck. A classic example of such a machine instruction format is the double-operand instruction of the PDP11. The instruction as stored in memory is a 16-bit value. These 16 bits are divided in 6 fields:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Opcode** | **Mode** | **Source** | **Mode** | **Destination** |
| 4 bits | 3 bits | 3 bits | 3 bits | 3 bits |

A simplified interpretation is that the Source and Destination each identify one of the 8 registers in the CPU, the Mode specifies how the register is to be used, and the Opcode specifies what is to be done. The instruction that adds the word value addressed by the register R2 to the content of register R3 (and stores the result in register R3) is encoded as shown in the next table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Opcode** | **Mode** | **Source** | **Mode** | **Destination** |
| 0110 | 001 | 010 | 000 | 011 |

Written as hexadecimal value we get 0x6283. This value gives very little information about what the instruction does. In the days of this processor octal notation was popular, and because the fields are 3 bits wide the octal notation is a bit more informative: 0o061203. The 5 fields of the instruction can immediately be recognized. (The Opcode field takes two positions.) There were people that could actually read and write programs for this processor in octal notation!

## Assembly language

Reading and writing PDP11 instructions in octal notation is possible, but it is very error prone, and other architectures do not have an equally regular division of bit fields. But we can make our life a lot easier by writing a ‘textual’ equivalent for the instruction, and have the computer translate this to the equivalent bits. This so-called assembly notation for our instruction would be

|  |
| --- |
| ADD [ R2 ], R3 |

ADD is the name of the instruction. R2 is the register that provides one of the inputs for the addition. The [ ] around R2 indicate that R2 is used as pointer to the data. Finally R3 is the other operand, which both supplies the second input for the addition, and is also the destination of the result. This notation uses a number of conventions, like left-to-right processing, and [ ] for a memory reference, that are unfortunately not standardized. Each processor has not just its own instruction set, but on top of that its own conventions for the textual equivalents. For another processor the instruction can for instance work from right to left, and use ( ) instead of [ ]. The only thing that seems to be common to all assembly languages is that the mnemonic (the word that states what the instruction does) comes first.

## Data path size

A fundamental choice a CPU designer must make is the size of the data it will work on. In most cases this will determine how many bits a register must store, the ALU must operate on, and the busses (both inside the CPU and the external bus between memory, CPU and I/O) must carry. If possible, this should also be the size of the machine instruction. A larger bit size means more work can be done by an instruction, but it also means more hardware (the CPU will occupy more die area). Historically there have been all sorts of weird sizes, like 4, 12, 14, 20 or 24 bits, but nowadays the common sizes are 8, 16, 32, and 64 bits. In general, a larger bit size makes a program run faster when larger numbers must be handled, but it does not help for handling smaller numbers. The next table shows the impact this has on things a CPU with a certain bit size can easily handle.

|  |  |  |
| --- | --- | --- |
| **Bit size** | **Maximum (unsigned) integer** | **Can (easily) handle** |
| 4 | 16 | One decimal digit |
| 8 | 255 | Small arrays, ASCII characters |
| 16 | 65535 | A/D converter results: sound, temperature, etc. |
| 32 | 4294967295 | Memory addresses up to 4 Gb |
| 64 | 6.2 x 1057 | Almost anything you can imagine |

## Instruction set design

The designer of an instruction format for a processor is faced with a number of dilemmas. One dilemma is the choice between wide and regularly divided instructions (easier for the CPU to decode) versus small but less regular instructions (can be delivered faster by the memory). Another dilemma is that a single instruction should be able to do a lot of work, because then for a given rate of instructions-executed-per-second the most work can be done. On the other side the instructions must be simple, because simple instructions can be executed faster. (Can you spot the fallacy in each reasoning?) The first approach culminates in the CISC (Complex Instruction Set Computer) style, the second approach in the RISC (Reduced Instruction Set Computer) style. Both have their merits, but the general trend in modern processors is towards RISC. Modern Intel processors, as can be found on almost every desk and lap, are CISC externally because they must be code-compatible with earlier CISC processors (from the days that CISC was the dominant style), but internally they translate these CISC instructions to one or more RISC instructions, because those can be executed so much faster that this seemingly clumsy approach is faster than direct execution of the original CISC instructions.

In the days that computers were programmed in assembly language it was important that it was easy for a programmer to write assembly langue. Gradually the emphasis shifted to the use of compiled high-level, and it became important that it was easy to write a compiler that translated a high-level language program into assembly. Both situations favored a CISC style processor, which can do a lot of work in one assembly instruction. Recently programming directly in assembly language is frowned upon, and the theory and practice of compiler writing has advanced to the point that it does not matter much what the instruction set of a processor looks like. If it can run the equivalent of a high-level statement fast, the compiler will find that way, much more reliable than a human can.

One choice an instruction set designer faces is how the operands for a calculation are specified. If the programmer can specify are all three (two inputs, one output) independently, the format is called a three-operand format. Specifying three operands in each instruction claims a lot of scarce bits. We can reduce this a bit by assuming that the result is stored in one of the operands. This limits the assembler programmer or compiler in what he can write or generate, but the reduced size of the instruction seems to outweigh this disadvantage.

A further limitation could be that the one operand that is both an input and the destination of the result must always be one fixed register, commonly called the accumulator. Such architecture is called an accumulator-based or one-operand architecture. We can take this even further, and require that all operands (the two inputs and the destination) are fixed. We could select fixed registers for this purpose, but a more common choice was to have the operands on the stack. This architecture is called stack-based or zero-operands.

Another aspect in which instruction sets differ is the options that are offered to a programmer for specifying an operand. Clearly the stack-based or zero-operand architecture is the most limiting in this respect: the programmer has nothing to choose. For the architectures that require (or allow) the programmer to specify an operand, the most limited form is to require that an operand is a register. This is called a register-register architecture, because the input for a calculation comes from registers, and the result is stored in a register. A number of bits from the instruction are used to identify which register is used for the result, which one as operand 1, and (for the three-operand format) which one is operand 2. On the one hand, a programmer would like to have a lot of registers available to make his work easier. But on the other hand the bits that specify a register take up scarce space in the instruction (the rate at which the memory can deliver instructions to the processor is limited), and registers themselves take up valuable space in the processor chip, so the instruction set designer and the chip designer want to limit the number of registers. Historically there have been mainstream architectures with the number of registers varying from 0 to 32.

Which architecture is optimal depends on factors that have changed over time, in particular the relative speed and cost of storing information in the processor itself, or storing it in main memory. Modern architectures are often register-based with 8 or 16 registers, using 16 bit instructions, and 32 or 64 bits data paths.

## Memory use

When an application is running it uses memory for a number of purposes. The common ones are shown in the next table. The area of memory used for one of these purposes is commonly called a region or segment.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Segment / Region** | **Used …** | **Present in executable** | **Allocated at** | **Allowed operations** |
| CODE | To store executable code | Yes | Build | Execute |
| RODATA | To store read-only data | Yes | Build | Read |
| DATA | To store data that has an initial value, but can be changed | Yes | Build | Read, write |
| BSS | To store data that has no specific initial value | No | Build | Read, write |
| STACK | For or the stack (local data and return addresses) | No | Run | Read, write |
| HEAP | For the heap (dynamically allocated data) | No | Run | Read, Write |

The CODE segment contains the executable instructions. It is often the largest part of an executable. When possible, during execution this region of will be protected from read and write operations, and it will be the only region that allows execution.

The RODATA segment contains data that will only be read. In C this corresponds to global const variables and literal constant values, including string constants. When possible, during execution this region will be protected from writing.

The DATA segment contains data that has an initial value, but can be overwritten by the running application. In C this is the global non-const data that has an initial value.

The BSS segment is for the global data that is not const, and has no specific initial value. The C and C++ languages specify that all bytes in this memory area are initialized to 0. Note that this region is not present in an executable: the executable only specifies how large this region is. The name BSS comes from Block Started by Symbol, but that has little to with its current use.

The STACK segment is used while the application runs, to store activation records of function or subroutine activations. In the C context an activation record contains the parameters, the return address, the local variables, and the function result. In general, the executable contains no explicit information about the stack segment, or even its size. When running, a hopefully large enough chunk of memory is allocated for the stack.

The HEAP segment is used when the application runs, to provide memory allocated my malloc (C) or new (C++) calls (and returned by free (C) or delete (C++) calls).[[1]](#footnote-1) As for the stack, in general the executable contains no explicit information about the heap or its size.

For the CODE, RODTA, DATA and BSS segments the linker decides where each code or data items is placed within each segment. The STACK and HEAP segments are managed at run time.

On a desktop system an executable is stored on disk. For running it is load into RAM, which is used via a virtual memory system. When an application is started, the contents executable is loaded in RAM, the BSS segment is allocated and cleared (all bytes are set to 0), and initial room for the STACK and HEAP segments is allocated. The protection for each segment (execute, read write) is set, and the application code starts executing. The STACK and HEAP segments are placed in memory in a way that allows these regions to be extended when they need more room.

|  |  |  |  |
| --- | --- | --- | --- |
| Executable file |  |  | RAM |
| CODE |  |  | CODE |
| RODATA |  |  | RODATA |
| DATA |  |  | DATA |
|  |  |  | BSS |
|  |  |  | STACK |
|  |  |  |  |
|  |  |  |  |
|  |  |  | HEAP |
|  |  |  |  |

On a microcontroller the application is stored in Flash ROM. Most microcontrollers have more ROM than RAM, so whenever possible the ROM content is used directly. Hence the CODE and RODATA segments are not copied to RAM. The DATA segment however must be copied, because it has to be writeable.

The BSS segment is allocated the amount of memory it needs. The remaining RAM is for the STACK and HEAP segments, which start to use this memory from opposite ends. This way all RAM can be used, even when it is not known in advance how much will be used by the two segments.

|  |  |  |
| --- | --- | --- |
| ROM |  | RAM |
| CODE |  | DATA |
| RODATA |  | BSS |
| DATA |  | STACK |
|  |  |  |
|  |  |  |
|  |  | HEAP |

# A bit of history

## Acorn computers to Arm Ltd.

In the 1980’s home computers was a booming business, but every manufacturer made its own computer, incompatible with all others. In the UK the BBC ran a popular educational series, using the computer affectionately called ‘the beep’: the BCC Micro, manufacturer by Acorn Computers. When the series ran to an end Acorn started development of a successor. This successor should have a GUI (Graphical User Interface), which required a much more powerful processor. At that moment the micro-processor market was dominated by 8-bit chips, with some 16-bit designs on the market, and 32-bit designs just emerging. Acorn evaluated a few 16-bit and 32-bit chips, but found them far too slow. In a bold move they decided to develop their own processor. As the existing 16-bit products did not offer enough performance, they went for a 32-bit design. With the up to that point in time dominant CISC philosophy (make the chip fast by making each machine instruction do a lot of work) that would have been totally impossible for a small company, but a new philosophy had recently emerged in the USA: RISC (make each machine instruction so simple that it can run very fast). The two chip designers (a ridiculously small team compared to contemporary efforts) had very little resources, and the company could not afford to develop a ‘big’ chip anyway. These restraints led to a very elegant 32 bit processor design: the ARM (Acorn’s Advanced RISC Machine). The computer products developed around this processor were not the commercial success the company had hoped for. The IBM PC became the dominant architecture, and all other designs (except for Apple’s products) vanished from the home computer market.

This could have been the end of the ARM design, but a new market was slowly emerging: handheld computers, which differed from desktop systems on two crucial points: there was no standard architecture and hence no existing set of software, and power consumption was (and still is) critical. The first meant that no architecture had an advantage over the others, and the second strongly favored the ARM design, which had almost by accident been a small and hence low-power design. Apple was actively exploring this new market, and they used ARM chips. The ARM design part of Acorn was spun off into ARM Ltd., which fully specialized in designing ARM chips. Unlike nearly all other ‘processor’ companies ARM Ltd. did not manufacture its designs, but instead licensed the designs to chip manufacturers. It still does so today.

## The low-power market

Meanwhile the IBM PC clones / Intel / Microsoft combination grew to be THE dominant force for desktop computers, at the expense of all other platforms (except, to some extent, Apple). But new markets for microcomputers emerged, and the ARM designs did very well in products like mobile phones, TV decoders, modems and routers. It is probably no coincidence that a lot of those devices run a version of Linux.

When the first ARM designs were made memory was generally faster than the processor, so it made sense to get a ‘big’ instruction with a fixed division into bit fields from memory to make it easy for the processor to execute it. Over time processors got much faster. Memory also got faster, but not nearly as much. This created a problem, especially in microcontrollers, a market in which the ARM had become one of the major architectures. A microcontroller integrates the processor, memory (Flash and RAM) and some peripherals all on one chip. This means that there is little or no room for a cache, which could bridge the speed gap between the processor and the Flash memory, so the processor’s speed is limited by the rate at which the memory can supply instructions. Another problem was that in a microcontroller the Flash (which stores the code) is a large part of the die area, and hence of the price of the chip. A solution to these problems that ARM (and the other designers of 32-bit processors of that time) found was to add a 16-bit instruction set to their 32-bit designs. A 16-bit instruction is of course less ‘powerful’ than a 32-bit instruction, but if you can get and execute them twice as fast they don’t need to be. And you can store twice as many instructions in the same Flash area! The 16-bit instruction set added to the ARM architecture were called Thumb. The resulting chips could execute both 32-bit and 16-bit instructions, and a program could consist of a mixture, because some types of programs were still better off (faster) when coded as 32-bit instructions.

This mixed 16/32 bit processor design could use the Flash memory very efficiently, but it was a bit of a mess: two instruction sets (hence a larger CPU, and more power consumption), a clever but complicated way to switch between the two, and a few seldom-used (but important) operations that could only be done by 32-bit instructions.

## From 32 bits ARM to 16 bits Cortex

Up to now all ARM designs were more-or-less code compatible with the original ARM chip: each new chip could execute the original instructions, and some more. With the Cortex family the ARM company broke with this tradition. The instruction set of the Cortex chips is the Thumb instruction set with extensions, but most of the original 32-bit ARM instruction set is gone. The result is an efficient instruction set, consisting of mostly 16-bit instructions.[[2]](#footnote-2) This instruction set is primarily designed to be used by compilers, not by human programmers, so it is much less regular than the original ARM 32-bit instruction set.

The Cortex is a family of three main designs: the Cortex M, R, and A. The Cortex M is the smallest design, optimized for a small die size and low current consumption. It is used in low-end microcontrollers. The Cortex R is optimized for Digital Signal Processing. Cortex A is optimized for high performance. This is the architecture you will likely find in mobile phones and other devices, often running Linux or a similar full-blown operating system. The instruction set of the Cortex M is a subset of the Cortex R, which is in turn a subset of the Cortex A. Within each category there are variants which are often denoted by a digit after the letter. This document focusses on the Cortex-M0 instruction set, which is the smallest of all Cortex instruction sets.[[3]](#footnote-3)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Cortex A instruction set | | |  |  |  |
|  | Cortex R instruction set | | |  |  |
|  |  | Cortex M instruction set | |  |  |
|  |  | | | |  |
|  | | | | | |

# Cortex M0 programming model

## Registers

The Cortex M0 is a 32-bit architecture: its registers and data paths are 32 bits wide, and the instructions operate on (up to) 32 bits of data. It has 16 registers (R0 ... R15), plus a second stack pointer (only one stack pointer is be active at a time), and a status register.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | R0 |  |  | R8 |  |  |
|  | R1 |  | R9 |  |
|  | R2 |  | R10 | High registers |
|  | R3 | Low registers | R11 |  |
|  | R4 | R12 |  |
|  | R5 |  | R13 = SP |  |  |
|  | R6 |  | R14 = LR |  |  |
|  | R7 |  | R15 = PC |  |  |
|  |  |  |  |  |  |  |
|  | status |  |  | SP |  |  |

The SP, LR and PC each have their specific use:

* PC (R15) is the Program Counter, which holds the memory address where the next instruction will be fetched. After fetching it is incremented by the size of the instruction (in most cases two), so it points to the next instruction.
* SP (R13) is the Stack Pointer, which is used by the PUSH and POP instructions to save registers on the stack and restore them later.
* LR (R14) is the link register, which is used to hold the return address when a subroutine is called.

The low registers are the real ‘general purpose registers’. The high registers are much less useful than the low registers; about the only thing you can do with the high registers is copying the content of one register to another.

The status register contains flags (Zero, Carry, Negative, Overflow) that reflect the result of a calculation, and can be used by a conditional branch (to implement an C/C++ if or looping construct).

There are two Stack Pointer registers: the Main Stack Pointer (MSP) and the Process Stack Pointer (PSP). A special register (CONTROL) contains one bit that determines which of the two stack pointers is used. When a Cortex M0 starts the MSP is used. An Operating System with memory protection could use MSP for itself, and the PSP for a user process. On a Cortex M0 the MSP is often the only stack used.

## Pipelining

To execute an instruction, a processor must go through a number of steps, for instance:

* Fetch the instruction
* Decode the instruction
* Fetch the required data (from registers, or from memory)
* Perform the operation on the data (calculate the result)
* Store the result

These operations are generally performed by different parts of the processor hardware. Hence when an instruction is for instance being in the process of calculating the result, large parts of the processor are idle (doing nothing). This is a waste, so most modern processors are pipelined: the processor hardware is a sequence of stages, that can each one of the tasks required to execute an instruction. This can dramatically increase the speed the number of instructions that can be executed. For a single-stage design each instruction must ripple through the full fetch, decode and execute circuitry, and only after that a new instruction can be put into the processor. Hence the maximum clock frequency is 1 / ( the\_total\_delay ). A pipelined design divides the circuitry in stages, with a storage register between the stages. Now each stage can be working on a different instruction. Now a new instruction can be put into the first stage (and each partially handled instruction shifted on to the next stage) when the slowest of the stages has completed its work. Ideally each stage takes the same time, so for an ideal N stage design instructions can be started N times as fast as for a single-stage design. (But note that each instruction still takes the same time from start to completion.) It might seem a good idea to make N (the number of stages) very large, but in practice it gets progressively more difficult to divide a processor as N increases, and the registers between the stages add an extra delay.

The Cortex M0 has a three-stage pipeline. The three stages are FETCH, DECODE, and EXECUTE. Each instruction progresses through these stages.

|  |  |  |
| --- | --- | --- |
| FETCH | DECODE | EXECUTE |

In the optimal situation each stage is working on a different instruction, and each clock cycle (tick of the clock) completes one instruction and starts another, hence the processor executes instructions at a rate of one instruction per clock cycle. Each memory access (which can be 4 bytes, 2 bytes, or 1 byte) adds an extra clock cycle.

|  |  |  |  |
| --- | --- | --- | --- |
| **Time 🡺** | FETCH | DECODE | EXECUTE |
| instruction 1 |  |  |
| instruction 2 | instruction 1 |  |
| instruction 3 | instruction 2 | instruction 1 |
| instruction 4 | instruction 3 | instruction 2 |
| **Progress through the CPU 🡺** | | |

Any instruction that modifies the PC (which happens in the EXECUTE stage) leaves the processor with two stages (FETCH and DECODE) that are working on the next two instructions that would have been executed when the PC had not been changed, but this work is now useless. The consequence is that modifying the PC effectively takes two extra cycles (hence a branch instruction counts as 3 cycles).

|  |  |
| --- | --- |
|  | **MOV**  **ADD**  **B brach\_to\_here** |
|  | **EOR**  **LSL** |
| **branch\_to\_here:** | **. . .**  **MUL**  **CMP**  **POP** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Time 🡺** | FETCH | DECODE | EXECUTE |
| B | ADD | MOV |
| EOR | B | ADD |
| LSL | EOR | B |
| MUL | idle | idle |
| CMP | MUL | idle |
| POP | CMP | MUL |
| **Progress through the CPU 🡺** | | |

In the fetch stage, after outputting the PC content to the address bus to read the instruction, the PC is incremented by two to point to the next instruction. A consequence of this is that when an instruction reads the content the PC (which it will do so in the execute phase), the PC has been incremented by two twice, so the value read is the address of the instruction plus four.

The Cortex-M0+ is a recent update of the Cortex-M0 architecture, with a 2-stage (fetch, execute) pipeline. Hence the branch penalty on this 1 clock cycle (a branch counts as 2 cycles).

The cycle counts as presented here assume that the memory bus is always free for fetching the next instruction. When the instruction reads or writes memory this is not the case: the memory interface will be busy doing this, so a fetch cycle that also wants to access the memory will be postponed for one cycle. This can be a major performance bottleneck, so solutions have been created.

A single 32-bit word contains two 16-bit Cortex instructions. The word read from memory is buffered in the processor, and if the next instruction is in the remaining 16 bits no extra memory access is needed. On top of this, reads from Flash memory often read more than one word (for instance 128 bits = 4 words = 16 Cortex instructions) so even less memory cycles might be needed.

Another solution is to have multiple memory busses. Instructions are mostly read from Flash and data is mostly read from and written to RAM, so it makes sense to have separate memory interfaces for these two memories. Higher end processors sometimes have RAM memory that has more than one interface, so it can handle more than one memory access at a time. Yet another solution is to divide RAM in sections, each with an independent memory interface. This is much cheaper than memory with multiple interfaces, but it requires that the programmer locates his data in RAM in an appropriate way.

# Cortex M0 assembly language

The Cortex M0 has a register-register instruction set: all operations are done on values in registers, and the result is written back to a register. In other words: the instructions that do calculations are different from the instructions that access memory. When memory is accessed, the default is to load or store a full (32 bit) word. The hardware can only do this from (or to) a four-byte aligned location (from an address that is divisible by 4). Variations of the load and store instructions work on a half-word (16 bits) or byte (8 bits). A half-word load or store must use an address that is two-byte aligned (address divisible by 2).

## Literal values

A literal value (also called ‘immediate’) is a value that can be calculated by the assembler. When this text shows N or M or #immedX[[4]](#footnote-4) such a literal value is expected.

Literal values include:

* Constant values, in binary (start with 0b), octal (start with 0), hexadecimal (start with 0x) or decimal (start with another digit)
* ASCII values, like ’A’
* Expressions formed from literal values and operators like +, -, \*, /
* In some contexts: labels

## Directives

An assembler source contains lines that produce values for memory locations, and lines that determine for which memory segment values are produced. The effect of the .text, .data and .bss directives is that subsequent values are to be placed in the CODE, DATA or BSS segments.

The assembler can handle machine instructions for a range of chips. To inform the assembler which chip we are using (so it can generate an error when we try to use an instruction that is not available on our chip) the directive .cpu is used. Cortex M0 Instructions must be 2-byte aligned. The directive .align 2 skips the number of bytes needed (zero or one) to get to an even address. This directive (with the value 4) is also used to get to a 4-byte aligned address before a memory word is allocated.

The .skip directive allocates a number of bytes in memory. Optionally you can specify the fill value for these bytes. The .byte, .hword and .word directives produce bytes, half-words or words in memory, filled with each of the indicated values. Note that you must arrange for these values to be properly aligned (with the .align directive)

|  |  |
| --- | --- |
| **Directive** | **Effect** |
| .align N | Skip addresses, until the address is divisible by N. |
| .ascii ”text”, ”text2”, … | Produce the bytes that contain the ASCII characters in each string. |
| .asciz ”text”, ”text2”, … | As .ascii, but add a ’\0’ byte after each string. |
| .bss | Following items are produced in the BSS segment (read/write, 0-initialized). |
| .byte N, M, … | Produce a number of bytes, each with one of the specified values. |
| .cpu cortex-m0 | Specify that the target CPU is a Cortex-M0. |
| .data | Following items are produced in the DATA segment (read/write, initialized). |
| .global Name, Name2, … | Make the indicated local labels accessible to other source files. |
| .hword N, M, … | Produce a number of half-words, each with one of the specified values. |
| .word N, M, … | Produce a number of words, each with one of the specified values. |
| .text | Following items are produced in the CODE segment (read-only). |

## Load, store and register-to-register move instructions

A surprisingly large amount of time a processor is shuffling data around, either between registers, or between registers and memory. The MOV instruction is used to move data from any register to any other register. The MOV name is misleading: the content of the source register is left intact, so COPY would have been a more appropriate name. The MOV instruction can also be used to load an 8-bit constant (range 0 ... 255) into a register.

The LDR instruction can be used to load a register with a value that is read from a memory location that is the sum of the contents of two other registers. All involved register must be low registers. In both cases the default is to load a full word (32 bits, 4 bytes). An H can be added to the mnemonic to load a half-word, a B can be added to load a single byte (8 bits). An alternative is to replace the second register with a 5-bit constant. For the default (load word) the 5-bit value in the opcode is multiplied by 4, for a half-word load by 2.

A special variant of the LDR instruction loads the register with the value at the address that is the sum of the PC (program counter) + 4 + an 8 bit immediate value. The value in the instruction is first multiplied by 4, so the range is 0 ... 1024 in steps of 4. In assembler the value we write down is the effective value (after the multiplication), not the value encoded in the bits in the machine instruction.

This instruction is used to ‘fake’ an additional format of the LDR instruction: loading a 32 bit contents value into a register. This can’t be done by a single instruction all by itself (why not?). Instead the PC-with-offset format is used, and the 32-bit value is stored ‘somewhere near’ the current instruction. Such 32-bit values are grouped together in what are called ‘constant pools’. An example:

|  |
| --- |
| **0x100: LDR R2, [ PC + 16 ] ; original instruction was: LDR R2, =0x1234**  **. . .**  **0x120: 0x1234** |

The STR instruction stores a value from a register in a location in memory. The STR instruction supports the same formats as the LDR instruction, except for the ‘fake’ form that loads a 32-bit constant value. (Why isn’t this form supported?)

The memory addresses used in load and store instructions must be aligned: a word address must be dividable by 4, a half-word address must be even (dividable by 2). If not, the processor will enter a fault state.

|  |  |
| --- | --- |
| **Format** | **Effect** |
| MOV Rd, Rm | Copy the content of Rm to Rd |
| MOV Rd, #immed8 | Load the value immed8 into Rd |
| LDR1 Rd, [ Rn, Rm ] | Load Rd with the content of the memory address Rn + Rm |
| LDR1 Rd, [ Rn, #immed52 ] | Load Rd with the content of memory address Rn + immed52 |
| LDR Rd, [ PC3, #immed8 ] | Load Rd with the content of memory address PC + 4 + immed8 \* 4 |
| STR1 Rd, [ Rn, Rm ] | Store the content of Rd in memory at address Rn + Rm |
| STR1 Rd, [ Rn, #immed52 ] | Store the content of Rd in memory at address Rn + immed52 |
| STR Rd, [ PC3, #immed8 ] | Store the content of Rd in memory at address PC + 4 + immed8 \* 4 |
| LDR Rd, =immed32 | Load Rd with the value immed32 |
| 1Append H for a half-word (16-bit) access, B for a byte (8-bit) access.  2The immed5 value in the opcode is multiplied by 4 for a word access, by 2 for a half-word access.  3Can also be SP, which is useful for compiled languages for accessing local variables the stack. | |

After the LDR and STR mnemonics an H or B can be put for half-word (16-bit) or byte (8-bit) use. When a half-word or byte is saved, the remaining bits of the register are simply ignored. When a half-word or byte is loaded into a register, the remaining bits are set to zero. This is appropriate for loading an unsigned value. For loading a 2’s-complement value a variant exists that sign-extends the loaded value, this preserving the value when interpreted in 2’s-complement format.

|  |  |
| --- | --- |
| **Format** | **Effect** |
| LDRSH Rd, [ Rn, Rm ] | Load Rd with the sign-extended half-word content of the memory address Rn + Rm |
| LDRSB Rd, [ Rn, Rm ] | Load Rd with the sign-extended byte content of the memory address Rn + Rm |

The stack instructions are used to save and later restore the content of registers on the stack. The convention is that a called subroutine does this for all registers it uses that are defined as ‘preserved’ by the calling convention. Register R13 is the stack pointer. A push first subtracts 4 from the stack pointer, and then stores the register that is pushed at that address. A pop first retrieves the register value from the address the SP was pointing to, and then it increments the SP by 4. This type of stack is called a FULL DESCENDING stack. Full means that the SP always point to an already occupied memory location. Descending means that the stack grows downwards (from high memory addresses to low memory addresses). There are three other types of stacks, which can be found on other architectures. For normal use (using PUSH and POP instructions) this does not matter much, but it does matter for initializing the SP and for using the SP to as a base address to access data stored on the stack. The actual PUSH and POP instructions allow for the specification of a list of (low) registers that must be pushed or popped. Additionally, a PUSH can also push R14 (the link register), and a POP can additionally pop the PC. Note that there is no direct way to push or pop the other high registers. If this is needed, it must be done via the low registers.

|  |  |
| --- | --- |
| **Format** | **Effect** |
| PUSH { low\_register\_list1 } | Push the indicated registers onto the stack |
| PUSH { low\_register\_list1, LR } | Push the indicated registers and the LR onto the stack |
| POP { low\_register\_list1 } | Pop the indicated registers from the stack |
| POP { low\_register\_list1, PC } | Pop the indicated registers and the PC from the stack |
| 1List of low registers, separated by commas, for instance ”R2, R3, R4, R6”. A range can be specified, for instance ”R2 – R4, R6”. | |

The load and store multiple instructions load or store any subset of the low registers to a memory address specified by a low register.

|  |  |
| --- | --- |
| **Format** | **Effect** |
| LDM Rn, { low\_register\_list } | Load the indicated registers from the memory pointer to by Rn and subsequent memory locations. |
| LDMIA Rn!, { low\_register\_list } | Load the indicated registers from the memory pointer to by Rn and subsequent memory location, then update Rn to point to the next higher memory address. |
| STMIA Rn!, { low\_register\_list } | Save the indicated registers from the memory pointer to by Rn and subsequent memory location, then update Rn to point to the next higher memory address. |

Two instructions read and write the special registers, like the status register, exception register, and control register. These instructions are unlikely to be used in most application programs.

|  |  |
| --- | --- |
| **Format** | **Effect** |
| MRS Rd, special-register | Read from a special register to a low register |
| MSR special-register, Rn | Write from a low register to a special register |

## Calculations

All arithmetic and logical instructions can be done on two registers (one in some cases), and the result can be stored in another register (bus the compare instructions do not store a result). The registers can be chosen freely from the low registers. All arithmetic and logical instructions update the status flags.

|  |  |
| --- | --- |
| **Arithmetic instructions** | |
| **Format** | **Effect** |
| ADD1 Rd, Rn, Rm | Rd = Rn + Rm |
| ADD Rd, Rn, #immed3 | Rd = Rn + immed3 |
| ADD Rd, Rd, #immed8 | Rd = Rd + immed8 |
| ADC Rd, Rn, Rm | Rd = Rn + Rm + Carry |
| SUB Rd, Rn, Rm | Rd = Rn - Rm |
| SUB Rd, Rn, #immed3 | Rd = Rn - immed3 |
| SUB Rd, Rd, #immed8 | Rd = Rd - immed8 |
| SBC Rd, Rn, Rm | Rd = Rn - Rm - Carry |
| MUL Rd, Rn, Rm | Rd = Rn \* Rm 2 |
| 1 For ADD only, all registers are allowed  2 The lower 32 bits of the result are kept, the higher 32 bits are ignored | |

|  |  |
| --- | --- |
| **Logical instructions** | |
| **Format** | **Effect** |
| AND Rd, Rn, Rm | Rd = Rn & Rm |
| EOR Rd, Rn, Rm | Rd = Rn | Rm |
| ORR Rd, Rn, Rm | Rd = Rn ^ Rm |
| BIC Rd, Rn, Rm | Rd = Rn & ! Rm |
| MOVN Rd, Rn | Rd = ! Rn |
| All logical operations are bitwise: bit N in the results is affected only by bits N in the source(s). | |

The shift instructions can shift or rotate the value in a low register a specified number of positions and store the result in a low register. The number of positions for the shift can be specified by an immediate value (1 ... 31), or by the value in another low register.

The logical shift instructions will shift in 0’s. The Arithmetic Shift Right instruction duplicates the value in the highest bit, thereby preserving the sign of the value when it represents a two’s-complement value. For both, the C (Carry) flag will set to the last bit shifted out.

The ROtate Right instruction rotates the bits: any bits shifted out at the low end are fed back in at the high end. For this instruction the number of shifts can be specified only by a register, not by an immediate constant. Note that rotating works circular, for instance rotating by 1 has the same effect as rotating by 33. Hence the number of rotate steps is used modulo 32.

There is no ROL (Rotate Left) instruction: ROL N has the same effect as ROR (32 – N).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Logical Shift Right (LSR) | 0 |  | register |  | C |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Logical Shift Left (LSL) | C |  | register |  | 0 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Arithmetic Shift Right (ASR) |  | register |  | C |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Rotate (ROR) |  | register |  |  |

|  |  |
| --- | --- |
| **Shift instructions** | |
| **Format** | **Effect** |
| LSL Rd, Rn, Rm | Rd = Rn << Rm |
| LSL Rd, Rn, #immed5 | Rd = Rn << immed5 |
| LSR Rd, Rn, Rm | Rd = Rn >> Rm |
| LSR Rd, Rn, #immed5 | Rd = Rn >> immed5 |
| ASR Rd, Rn, Rm | Rd = Rn / 2 \*\* Rm |
| ASR Rd, Rn, #immed5 | Rd = Rn / 2 \*\* immed5 |
| ROR Rd, Rn, Rm | Rd = ( Rn >> ( Rm % 32 )) | ( Rn << ( 32 – ( Rm % 32 ))) |

The extend instructions are used when a register contains a value in its lower 8 or 16 bits, and you want to extend this to all 32 bits. For an unsigned value this means that the higher 24 or 16 bits must be filled with zero bits. For a 2’s-complement signed value this means that the highest valid bit must be copied into the higher 24 or 16 bits.

|  |  |
| --- | --- |
| **Extend instructions** | |
| **Format** | **Effect** |
| SXTH Rd, Rm | Sign-extend the half-word 2’s-complement value in Rm , store in Rd |
| SXTB Rd, Rm | Sign-extend the byte 2’s-complement value in Rm , store in Rd |
| UXTH Rd, Rm | Zero-extend the half-word unsigned value in Rm , store in Rd |
| UXTB Rd, Rm | Zero-extend the byte unsigned value in Rm , store in Rd |

The compare and test instructions calculate a value, but it is not stored. The only effect is that the status flags are updated according to the calculated value.

|  |  |
| --- | --- |
| **Compare and test instructions** | |
| **Format** | **Effect**1 |
| TST Rn, Rm | Calculate Rn & Rm |
| CMP Rn, Rm | Calculate Rn - Rm |
| CMN Rn, Rm | Calculate Rn + Rm |
| CMP Rn, #immed8 | Calculate Rn – immed82 |
| 2and the flags are set!  1range -128 … +127 | |

The stack is commonly used to store local variables that do not fit in the available registers. Three special forms of add and subtract instructions support this. The stack grows towards lower addresses. Hence the subtract instruction is used to claim an area of memory for local variables, and the add instruction is used to free it again. To calculate the address of a local variable and put it in a register, an immediate value is added to the stack pointer and the result is put in a low register.

|  |  |
| --- | --- |
| **Add and subtract SP** | |
| **Format** | **Effect** |
| ADD SP, SP, #immed71 | SP = SP + immed7 |
| SUB SP, SP, #immed71 | SP = SP - immed7 |
| ADD Rd, SP, #immed82 | Rd = SP + immed8 |
| 1range 0 ... 512 in steps of 4  2range 0 ... 1024 in steps of 4 | |

Three instructions are provided to convert a value from big endian to little endian or vice versa.

|  |  |
| --- | --- |
| **Endianness conversion** | |
| **Format** | **Effect** |
| REV Rd, Rm | Convert a little endian 32-bit value to big endian, or vice versa |
| REV16 Rd, Rm | Convert an unsigned little endian 16-bit value to big endian, or vice versa |
| REVSH Rd, Rm | Convert a signed little endian 16-bit value to big endian, or vice versa, and sign-extend into the higher 16 bits |

## Flow control instructions

When an instruction has been fetched the PC is incremented by two, so it points to the next instruction, which is fetched the next fetch phase. The flow control instructions change this pattern by writing a new value to the PC. Note that when this happens, two more instructions have already been fetched. These instructions will be discarded. The PC can be changed by an instruction that has the PC as destination of a move, pop, or addition.

|  |  |
| --- | --- |
| **Format** | **Effect** |
| MOV PC, Rm | Move any register to PC |
| POP { low\_register\_list, PC } | Pop the indicated registers and the PC from the stack |
| ADD PC, PC, Rm | Add low register to PC |

The branch instructions put a new value in the PC which is specified by a label. The actual instruction contains an offset that is added to the PC. For the plain branch instruction this offset is +/- 2046 bytes, for the conditional branch it is +/- 254 bytes. This might seem very limited, but branches are mostly within a subroutine, for which they are often sufficient. When the +/- 254 of a conditional branch is not sufficient, a plain branch can be used, with a conditional branch with the opposite condition that skips the plain branch.

|  |
| --- |
| **// when the label is too far away**  **// for the +/- 254 allowed by a conditional branch**  **BEQ far\_away\_label**  **// you can replace it with a conditional branch**  **// with the opposite condition**  **BNE do\_not\_branch**  **// that braches around a plain branch to the original label**  **B far\_away\_label**  **do\_not\_branch:** |

If even the range of an unconditional branch is not sufficient the target address can be loaded in a register followed by a MOV PC, Rm instruction.

|  |  |
| --- | --- |
| **Format** | **Effect** |
| B label | Branch to the label |
| B*cond*1 label | Branch to the label if (and only if) the condition holds |
| BX Rm2 | Branch to the address in the register |
| 1The conditional branch mnemonics can be one from the next table.  2The lowest bit of the register must be 1. | |

The conditional branch instruction mnemonics are formed from a B for branch, followed by two letters that specify the condition. The next table shows the conditional branch instructions and the condition that must hold for the branch to be taken. If the condition does not hold, execution continues with the next instruction (the instruction after the branch).

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Condition** | **Flags** |
| BCC / BCLO | Carry clear / unsigned lower | !C |
| BCS / BHS | Carry set / unsigned higher or same | C |
| BEQ | Equal | Z |
| BGE | Signed greater than or equal | N == V |
| BGT | Signed greater than | !Z & ( N == V ) |
| BHI | Unsigned higher | C & !Z |
| BLE | Signed less than or equal | Z | ( N != V ) |
| BLS | Unsigned lower or same | !C | Z |
| BLT | Signed less than | N != V |
| BMI | Minus / negative | N |
| BNE | Not equal | !Z |
| BPL | Plus / zero or positive | !N |
| BVC | No overflow | !V |
| BVS | Overflow | V |

The branch-and-link instructions are used to jump to a subroutine. These instructions first save the PC value in the LR, and then branch to the new address. When the subroutine is finished, it must copy the LR content back to the PC to resume executing at the instruction after the branch-and-link.

|  |  |
| --- | --- |
| **Format** | **Effect** |
| BL label | Save PC to LR, then branch to the label |
| BLX Rm1 | Save PC to LR, then branch to the address in the register |
| 1The lowest bit of the register must be 1. | |

Cortex instructions are either 16 or 32 bits, and must be half-word-aligned (hence the lowest bit of the address must be 0). In the previous generation of ARM processors this ‘unused’ bit was used to switch between the ARM and Thumb instruction sets. A 0 in this bit specified the ARM instruction set; a 1 specified the Thumb instruction set. The Cortex M0 does not implement the ARM instruction set, and it will go into a fault mode when a switch to the ARM instruction set is attempted. For the BX and BLX instructions (the ones for which the target address is in a register) the lowest bit of the address must be set (you must add 1 to the actual address).

## Miscellaneous instructions

The instructions in the next table are shown for completeness. They instructions are unlikely to be used in most application programs.

|  |  |
| --- | --- |
| **Format** | **Effect** |
| SVC #immed8 | Supervisor call |
| CPSID n | Disable interrupts |
| CPSIE n | Enable interrupts |
| Breakpoint #immed8 | Breakpoint, used by a debugger |
| SEV | Send event |
| WFE | Wait for an event |
| WFI | Wait for an interrupt |
| YIELD | Yield – executes as a NOP on a Cortex M0 |
| NOP | No operation – does nothing |
| ISB | Instruction synchronizations |
| DMB | Data memory barrier |
| DSB | Data synchronization memory |

# Subroutines

## Calling Standard

The ARM Procedure Call Standard defines how a subroutine is called, and especially the role of the various registers.

The first four registers, R0 … R3, are used to pass arguments to the subroutine. The first two are also used to pass a result (return value) back to the caller. In-between call and return the subroutine is free to use these registers as it sees fit, and it is not obliged to preserve the content of these registers.

When four registers are not enough to pass all parameters, R3 points to a piece of memory where the remaining argument values are stored.

The next eight registers, R4 … R11, are used for local variables. A called subroutine must preserve the values in these registers, hence when it uses one of these registers, it will save it value on entry to the subroutine, and restore it on exit.

The R12 registers is reserved for use by the linker. Whether it actually uses it depends on the tool chain (compiler and linker) you are using. I found no information on it being used (or not used) on GCC, so better treat it as reserved.

The last three registers, SP, LR and PC, each have a specific function.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Low  registers |  |  |  |  |
|  |  | R0 |  | Arguments,  Retrun value,  Scratch  (not preserved) |
|  | R1 |
|  | R2 |
|  | R3 |
|  | R4 |  | Local variables (preserved) |
|  | R5 |
|  | R6 |
|  | R7 |
|  |  |  | R8 |
|  |  |  | R9 |
|  |  |  | R10 |
|  |  |  | R11 |
|  |  |  | R12 |  | reserved |
|  |  |  | R13 |  | Stack Pointer |
|  |  |  | R14 |  | Link Register |
|  |  |  | R15 |  | Program Counter |
|  |  |  |  |  |  |

On a Cortex M0, the low registers are much easier to use, so a simple approach is to use only the low registers.

The BL or BLX instructions are used to call a subroutine. They both save the return address in the LR register. A subroutine that does not call any other subroutines and does not need any registers beyond the scratch registers can hence be very simple. As an example, consider a subroutine that returns the sum of its two 32-bit integer arguments. The C/C++ equivalent is shown beside the assembly code.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  | **int add( int a, int b ){**  **return a + b;**  **}** |  | **add:** | **ADD R0, R0, R1**  **MOV PC, LR** |

The two arguments are passed in R0 (int a) and R1 (int b). The result is expected in R0, so a single ADD instruction is all that is needed. The return address is in LR, so copying it to the PC will cause execution to resume after the subroutine call.

The next example prints a string by calling a print\_char function for each char in the string, until a \0 character is encountered. A pointer is used to ‘walk through’ the string in typical C style. A C++ programmer might prefer to use an index, but the effect after compilation will likely be the same.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  | **void print\_string( char \*s ){**  **while( \*s != 0 ){**  **print\_char( \*s );**  **s++;**  **}**  **}** |  | **print\_string:**  **loop:**  **done:** | **PUSH { R4, LR }**  **MOV R4, R0**  **LDRB R0, [ R4 ]**  **TST R0, #0**  **BEQ done**  **BL print\_char**  **ADD R4, R4, #1**  **B loop**  **POP { R4, PC }** |

This example is more complex because our subroutine is not a leaf subroutine: it must call another subroutine. This other subroutine is not required to preserve any of the argument / scratch registers, so the character pointer must be saved in a local variable register, in this case R4. But those local variable registers must be preserved by our subroutine, so we must save this register on entry to the subroutine, and restore it on exit. Additionally, calling another subroutine will overwrite the LR, so we must save and restore that register too. This can be simplified a little by restoring it not to the LR, but directly to the PC, which causes our subroutine to return.

To summarize, the normal use of the PUSH and POP instructions in a subroutine that is not a leaf (branches further to other subroutines) is

* At the start of a subroutine, PUSH all local variable registers that are changed in the subroutine, and the LR
* At the end of the subroutine, POP the same registers and the PC.

## Local variables

When a procedure, function or subroutine (at assembly level these are all the same) needs local variables, there are a number of places where they can be stored.

The easiest is to store them in the scratch registers (R0-R3), because this requires no further action at all. This is possible when there are scratch registers left (they are used for passing parameters), and our subroutine does not call any further subroutines (because those might corrupt the scratch registers – the same calling convention that allows our subroutine to use these registers freely allows another subroutine to use them too).

The next best option is to use the local variable registers (R4 - R7). When used, the calling convention requires that our subroutine stores such registers on entry, and restores them on exit. This is easily done with a PUSH and a POP. If our subroutine calls any further subroutines we need these PUSH and POP anyway to preserve the LR.

In some cases it might be convenient to use some of the high local variable registers (R8 - R11) too, but because most instructions can’t be used with these registers it is often not worth the trouble. When used, these high registers must be PUSHed and POPed too. (Which is complicated by the fact that the PUSH and POP instructions can’t use the high registers directly: they must be copied to a low register and PUSHed, or POPed to a low register and then copied to the high register.)

When coding a non-reentrant subroutine (one that is guaranteed never to be active more than once at the same time: not call itself, not even via a chain of other subroutines, and not used by more than one thread) global memory locations can be used as ‘local’ variables. This is relatively easy to do (allocate the memory with .word directives in the DATA or BSS segment). The disadvantage is that recursion and multithreading is not possible, and the data is permanently allocated.

A compiler will generally allocate local variables (that can’t be kept in registers) on the stack. On entry to a function, the SP is lowered to make room for the local variables, and at exit the SP is restored. The base address of the thus allocated memory is stored in a register that serves as Base Pointer. (Some CPU architectures have a dedicated Base Pointer register, but the Cortex does not, so it can use any convenient register. With some extra care, the Base Pointer register can even be omitted.)

The variables can now be addressed (LDR and STR instructions) by using the BP with an appropriate offset. The example below shows the code generated by GCC for a very simple function, without optimization (-O0).

|  |
| --- |
| **int f( int y ){**  **int x = y + 1;**  **return x + 3;**  **}** |

|  |
| --- |
| **f:**  **push {r7, lr} // save registers**  **sub sp, sp, #16 // make room for locals**  **add r7, sp, #0 // r7 is the Base Pointer**  **str r0, [r7, #4] // store the argument**  **ldr r3, [r7, #4] // x = y + 1 get y**  **add r3, r3, #1 // add 1**  **str r3, [r7, #12] // store x**  **ldr r3, [r7, #12] // calculate x + 3 get x**  **add r3, r3, #3 // add 3**  **mov r0, r3 // and return it**  **mov sp, r7 // restore the SP**  **add sp, sp, #16**  **pop {r7, pc} // restore registers and return** |

Note that the stack is lowered by 16 bytes, hence there is room for 4 word-sized variables. One (at offset 4) is used to save the argument, another (at offset 12) is the local variable x. The other two (offset 0 and 8) are not used, I have no idea why they are allocated.

The assembly code generated by the compiler without optimizations is instructive, but not typical for ‘production’ code. With the option to optimize for size (-Os) the compiler generates the code shown below. The two additions are folded into one, the local variable has disappeared, and the argument is not stored in memory. All that remains is a single addition and the return. Note that debugging this code might put you in for surprises: there is no local variable of which you can get or change the value, and the two source lines have been folded into one, so you can’t set a breakpoint on a specific line.

|  |
| --- |
| **f:**  **add r0, r0, #4**  **bx lr** |

## Combining C and assembly

In assembly, a label that is defined in a source file is by default local (only visible in that source file). To make it visible to other files, you must mention it in a .global directive.

A label that is used but not defined is automatically assumed to be provided by another file. Hence when you misspell a label the assembler will not warn you. Instead the linker will complain about a missing name. (That is, if you are lucky! If Murphy has his day the label will be defined and made global by one of the other files…)

In C the situation is the reverse. A C function is automatically available to other files. But when you want to call an assembler subroutine from C, you must tell the C compiler that a function of that name exist, and how it is to be called (with what parameters, and what it will return). The C compiler will pass the parameters (and use the return value) as specified by the calling standard.

|  |  |  |
| --- | --- | --- |
| Assembler calling C |  |  |
|  |  |  |
| **mov R0, #5**  **mov r1, #1**  **bl plus**  **// r0 is now 6** |  | **int plus( int x, int y ){**  **return x + y;**  **}** |

|  |  |  |
| --- | --- | --- |
| C calling assembler |  |  |
|  |  |  |
| **int plus( int x, in y );**  **int y = plus( 5, 1 );**  **// y is now 6** |  | **.global plus**  **add:**  **add r0, r0, r1**  **bx lr** |

In C++ the situation is much like C, except for the external name of a function. C++ has function overloading: there can be more than one function with the same name, distinguished by the parameters. To implement this with a linker that has no knowledge, the C++ compiler mangles the external name of each function by appending the names of the arguments. How exactly this is done is compiler-dependent, hence it is not a good idea to rely on it.

For interfacing with C and assembler C++ has the option to disable overloading and name mangling for a function. This is done by prefixing the function with extern "C". Use this if you must call a C++ function from assembler (or C), or call an assembler (or C) function from C++.

# Startup

When a Cortex starts up (after a power up, or a reset) the startup code takes control. It must do at least the following things:

* 1. The Stack Pointer must be initialized
  2. The memory set aside for the BSS must be cleared
  3. The copy of the DATA segment that is store in ROM must be copied to RAM
  4. The main must be called

When the main returns, control returns to the startup code. Probably the best it can do is entering a never-ending loop. (Other options could be to generate an error message – but how? -, or to re-start the application.)

A Cortex chip starts by loading its SP and PC from the first two words of memory. (The next 14 words are addresses of code that is executed when a special condition occurs.) In standard C we can’t locate something at a fixed address, but most compilers have a compiler-specific extension for this. In GCC we can use the \_\_attribute\_\_\_((section(”*section-name*”))) to place a something (in this case an initialized array) in a specific section (without the attribute it would end up somewhere in the RODATA section).

|  |
| --- |
| **void \_\_startup( void );**  **extern int \_\_stack\_end;**  **const int startup\_vectors[ 16 ]**  **attribute\_\_((section(".vectors"))) = {**  **(int) & \_\_stack\_end,**  **(int) \_\_startup**  **};** |

The first two entries of the 16-word table must be filled with the initial value for the SP, and the initial value for the PC. We could specify the initial SP by hand: it is the last RAM byte + 1. (Remember, the stack is customary allocated at the end of memory, and it always points to the lowest used byte.) Instead we leave it to the linker to determine the value. The linker script must know the RAM size, and this way startup code does not. The linker can supply a value ‘back’ to the C code as a label. We can ‘catch’ this value by defining something with the same name. It does not matter what it is (it does not actually exist, it only has an address). We initialize the first entry in the table with the address of the end-of-stack label. It must take the address (because it is an address) and then cast it to an int (because the table is an int array).

The second entry in the table is the initial PC. Here we place the address of a \_\_startup() function that we provide, which does the required initialization, and then calls main(). This function needs to know the start and end of the sections in memory. It gets this information from the linker in the form of a number of agreed-upon names.

The other entries in the startup vectors table point to the code that is executed when certain exceptions or interrupts occur. In this simple startup no such code is used.

The startup code first clears the BSS section. The linker provides the first and last-plus-one addresses of the section, and makes sure that both addresses are 4-byte aligned.

Next the startup code copies the initial values for the DATA section from the ROM to RAM. Note that the linker provides two start labels for the DATA section, one for the actual location in RAM, and one for the initial values in ROM. Both are of course the same length, so the end label of only one is needed.

When the BSS and DATA sections have been initialized the main() can be called. If the main() ever returns, the while(1) will keep the CPU occupied until the chip is reset.

|  |
| --- |
| **extern int \_\_data\_init\_start;**  **extern int \_\_stack\_start;**  **extern int \_\_stack\_end;**  **extern int \_\_data\_start;**  **extern int \_\_data\_end;**  **extern int \_\_bss\_start;**  **extern int \_\_bss\_end;**  **void \_\_startup( void ){**  **unsigned int \*s, \*d, \*e;**  **// clear .bss section**  **d = & \_\_bss\_start;**  **e = & \_\_bss\_end;**  **while( d != e ){**  **\*d++ = 0;**  **}**    **// copy .data section from flash to ram**  **s = & \_\_data\_init\_start;**  **d = & \_\_data\_start;**  **e = & \_\_data\_end;**  **while( d != e ){**  **\*d++ = \*s++;**  **}**    **// call main**  **(void) main();**    **// when main returns: loop till hell freezes down**  **while(1){}**  **}** |

# Building an application

When all sources have been compiled, the linker gathers the segments from all object (.o) files, and puts them in the right place in memory, according to a linker script. This script specifies in a rather bizarre syntax

* the properties of the target chip (how much RAM and ROM does it have, and at which addresses),
* the entry point of the application, and
* where the various sections from the .o files are to be placed.

The linker does not include all sections in the application. Its principle is that a section is used if at least one symbol provided by that section is used by another section that is already used: it includes sections on an as-needed base. Without any initial labels or sections the application would remain empty. There are two mechanisms that force a label or section into the image: the ENTRY specifies the entry point of the application, and section that provides that label is hence used, and sections marked as KEEP in the sections entries are always included. In our case the special section ”vectors” is specified as KEEP, and it needs the \_\_startup, which is also mentioned as ENTRY.

The following script is much simplified: there are far more special sections (mostly having to do with debugging and C++ ) than shown here.

|  |
| --- |
| **MEMORY {**  **rom (rx) : org = 0x00000000, len = 512k**  **ram (rwx) : org = 0x20000000, len = 96k**  **}**  **ENTRY( \_\_startup );**  **SECTIONS{**  **.text : {**  **KEEP(\*(.vectors))**  **. = ALIGN(4)**  **\*(.text .text.\* )**  **. = ALIGN(4)**  **\*(.rodata .rodata.\*)**  **} > rom AT > rom**  **.data : {**  **. = ALIGN(4)**  **\_\_data\_init\_start = LOADADDR (.data)**  **PROVIDE(\_\_data\_init\_start = \_\_data\_init\_start)**  **\_\_data\_start = .**  **PROVIDE(\_\_data\_start = \_\_data\_start)**  **\*(.data .data.\* )**  **. = ALIGN(4);**  **\_\_data\_end = .**  **PROVIDE(\_\_data\_end = \_\_data\_end)**  **} > ram AT > rom**  **.bss : {**  **. = ALIGN(4)**  **\_\_bss\_start = .**  **PROVIDE(\_\_bss\_start = \_\_bss\_start)**  **\*(.bss .bss.\*)**  **. = ALIGN(4)**  **\_\_bss\_end = .**  **PROVIDE(\_\_bss\_end = \_\_bss\_end)**  **} > ram AT > ram**  **.stack : {**  **. = ALIGN(8)**  **. += \_\_stack\_size**  **. = ALIGN(8)**  **\_\_stack\_end = .**  **PROVIDE(\_\_stack\_end = \_\_stack\_end)**  **} > ram AT > ram**  **}** |

Note the special handling of the DATA: the “> ram” places it in RAM (that’s where label the values in the section will refer to), but “AT > rom” makes that the data values itself are loaded in ROM.

The last segment is the stack, which simply aligns to an 8-byte boundary, reserves the amount of memory needed for the stack, aligns to 8-bytes again, and provides the current address to the code.

The output of the linker is the executable file. For windows the extension is .exe (or .com), for Linux the file has no extension. It contains the segments that have been discussed, but also information for the debugger. This information can be stripped, which can make a big difference in the size of the executable, but it has little influence on the running, because normally the debug information is not loaded when an application is run.

For a microcontroller the linker produces a file in a comparable format, often .elf. The next step is to extract what has to be put in the ROM, which is just the CODE and DATA segments, without any debug information. The file format for this ROM image is often .hex. Finally this .hex file is downloaded into the microcontroller’s ROM. For a SAM38XE this can be done by starting the chip in boot load mode, by pulling a specific pin low during startup. Now a dedicated non-writable ROM in the chip takes control, and communicates over the serial port with a suitable application on the PC, which transfers the content of the .hex file to the chip, where it is written to the FLASH ROM.

# Context switching

At the machine instruction level (which is roughly the same as the assembler level) a CPU will execute a sequence of instructions. A C or C++ compiler will translated the control flow of the high level code into such a sequence, so by default the CPU will execute what corresponds to one high level language control flow. Let’s call this a thread.

When a program must perform more than one task, and the timing of these tasks is independent, like ‘blink this LED at 33 Hz, this speaker at 3 kHz, and echo each character received over the UART’, it is much easier to write the program as a number of threads.

## Cooperative multithreading

When left to itself the CPU will execute only one thread of instructions, but we can fake that there are three CPU’s by switching the single CPU between the three threads at appropriate moments. The easy way is to do this when a thread has to wait anyway. This is called cooperative multitasking: the thread that waits cooperates by passing the CPU to another thread. To make this possible we must be able to save the thread context (everything that is associated with a thread) of the current thread, and load the context of the next thread that wants the CPU. This action is called a context switch.

In compiled code the context is on the stack (local variables, temporaries, return addresses) and in the registers. Hence each thread must have its own stack, and when a switch to another thread is made the registers must be saved on the stack and the stack pointer remembered. That stack pointer is the information we need later on to switch back to the saved context and resume execution. Such actions can’t be expressed in C/C++, but the interface can. The function gets two arguments: a pointer to the place where the old SP must be stored, and the value of the new SP. Note that this is a very strange function: execution enters the function while the CPU is executing one thread, but when it leaves it is executing another thread

|  |
| --- |
| **extern ”C” void switch\_from\_to(**  **int \*store\_old\_sp,**  **int next\_sp**  **);** |

However strange the effect of our switch\_from\_to function might be, it is still a function, which must adhere to the calling convention. Hence the single parameter is passed in R0, and registers R4 – R12 must be preserved to be restored later when the thread is resumed. First we push R4 – R7 and the LR, like any normal function that uses these registers would do. The high registers cannot be pushed directly, hence we copy them to low registers and push those.

Now that we have pushed everything our SP can be saved. (This can obviously not be done before the push instructions.) The SP is saved in the location pointed to by store\_old\_sp.

Next the SP is set to the value of next\_sp. Now the stack is the stack of the new thread, and we can pop the registers in the reverse order in which they were pushed.

|  |
| --- |
| **switch\_from\_to:**  **// save current context on the stack**  **push { r4 - r7, lr }**  **mov r2, r8**  **mov r3, r9**  **mov r4, r10**  **mov r5, r11**  **mov r6, r12**  **push { r2 - r6 }**    **// \*store\_old\_sp = SP**  **mov r2, sp**  **str r2, [ r0 ]**    **// SP = next\_sp**  **mov sp, r1**    **// restore the new context from the stack**  **pop { r2 - r6 }**  **mov r12, r6**  **mov r11, r5**  **mov r10, r4**  **mov r9, r3**  **mov r8, r2**  **pop { r4 - r7, pc }** |

Obviously, the context switch code assumes that the stack is in the state created by a previous context switch call, so once a thread is running and switches to another thread it can be switched back to. This leaves the problem of how a thread started in the first place. A simple way is to write a thread as a function (similar to a main()), and to construct the stack as switch\_from\_to assumes it to be. For this we need room in memory for the stack. A very simple way to do both is to extend the thread type with room for the stack. For a start, make it 400 bytes.

Each thread must be able to wait individually. To make this possible we add a wakeup moment to the thread structure, and a thread pointer that makes it possible to arrange the threads in a singly linked list.

|  |
| --- |
| **class thread;**  **thread \* first\_thread = NULL;**  **class thread {**  **public:**  **int \* sp;**  **int stack[ 100 ];**  **thread \* next;**  **int wakeup\_time;**  **} thread;** |

The switch\_from\_to routine pops 9 working registers and the PC from the stack. The stack of a Cortex is full descending: descending means that it grows from higher addresses to lower addresses, and full means that it points to memory that is NOT free. For our stack, pointing to stack[ 100 ] (beyond the memory we allocated!) would correspond to the stack being empty. Pushing 10 registers would have the stack pointer point to stack[ 100 – 10 ].

When we use the push and pop instructions we don’t care in which order the registers are pushed, but now we do care: we must arrange the stack so that the final pop instruction in the switch\_from\_to subroutine will pop the address of the threads ‘main’ function into the program counter. On careful reading documentation of the Cortex states that the PC will be the last register popped from the stack, hence it must be put at stack[ 100 – 1 ]. Except for the PC and SP we don’t care what ithe initial register contents is when the ‘main’ function of the thread starts, hence we don’t need to initialize the other stack entries.

The thread is not yet waiting, hence the wakeup\_time is set to zero, and finally the thread is inserted into the linked list of threads.

|  |
| --- |
| **void thread\_init( thread \* t, void f() ){**  **t->stack[ STACK\_SIZE - 1 ] = (int) f;**  **t->sp = (int) & t->stack[STACK\_SIZE – 10 ];**  **wakeup\_time = 0;**  **t->next = first\_thread;**  **first\_thread = t;**  **}** |

When a thread wants to wait for an amount of time it sets its wakeup\_time to the point in the future at which it want to return from the wait. Next it walks the list of threads to find a thread that is ready to run, that is: a thread for which the wakeup\_time has passed. Then it switches to that thread. The only service it needs to do this is a function that returns the current time.

|  |
| --- |
| **thread \* find\_thread(){**  **thread \* t;**  **for(;;){**  **for( t = first\_thread; t != 0; t = t->next ){**  **if( current\_time() >= t->wakeup\_time ){**  **return t;**  **}**  **}**  **}**  **}**  **void wait( int d ){**  **current\_thread->wakeup\_time = current\_time() + d;**  **switch\_to( find\_thread() );**  **}** |

To use our toy thread switcher we must first create the threads that we want to run, and then switch to the first thread. From then on the threads will (when they execute a wait) start the next thread. We could just do a switch\_to() on the head thread in the list, but for one detail: the switch\_to will (after saving the current register on the stack, which is not a problem) save the current stack pointer in the location pointed to by current\_thread. Hence we must make sure that current\_thread points somewhere sensible. We could allocate a full thread structure for this purpose, but that would be a waste or RAM, because only the sp part of it would ever be used. Hence we allocate just one integer, and force current\_thread to point to it.

|  |
| --- |
| **int main\_sp;**  **thread \* current\_thread = ( thread \*) & main\_sp;**  **void rtos(){**  **switch\_to( first\_thread );**  **}** |

Note that to make context switching possible, the code executed by the threads must be re-entrant: it must not use anything outside its context in way that would cause trouble when two or more copies of the code are running in parallel. The main cause of non-reentrant code is the use of shared resources, especially global variables. Code generated by the compiler will not use global variables, except when we explicitly do so in the high level source code.

## Preemptive multithreading

The toy threading system will switch from one thread to another only when the first thread executes a wait. This is called a cooperative threading system: each thread must cooperate to keep the system ‘alive’. A single thread that hogs the CPU forever kills all other threads.

A preemptive system does not solely rely on the cooperation of the threads, but also uses a timer interrupt to suspend a thread and possible switch to another one. This has a number of advantages:

* A single thread that needs the CPU for a long time will no longer block the execution of other (higher priority) threads.
* An external interrupt (for instance from the hardware UART, I2C or SPI) can be used to activate a thread. (A cooperative system would need to poll such hardware.)
* Multiple threads that have the same priority (for instance threads that serve users at interactive terminals) can share the CPU, so they can each provide a reasonable service to their user.

These benefits come at a cost. In a preemptive system thread switching can occur at after machine instruction (on modern chips even halfway the execution of a machine instruction). This can produce unexpected results. The example below runs two threads, one increments a 64-bit value, the other prints the value. On a 32-bit CPU incrementing a 64-bit value must be done in two steps, first the lower 32 bit, then the higher 32 bit. The two sub-results cannot be stored back to memory by one instruction; hence there will always be a moment in time where one of the sub-results has been stored, while the other hasn’t. When a thread switch occurs at that moment the other thread will print a strange value.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **uint64\_t int n = 0;**  **// thread 1**  **for(;;){**  **n++;**  **}**  **// thread 2**  **for(;;){**  **std::cout << n <<”\n”;**  **}** |  | **// n++**  **ldr r0, = n**  **ldr r1, [ r0 ]**  **add r1, r1, #1**  **str r1, [ r0 ]**  **ldr r1, [ r0, #4 ]**  **adc r1, r1, #0**  **str r1, [ r0 ]** |  | **. . .**  **0000 0000 FFFF FFFE**  **0000 0000 FFFF FFFF**  **0000 0000 0000 0000**  **0000 0001 0000 0000**  **0000 0002 0000 0001**  **. . .** |

Note that the problem originates from something that is shared between the two threads: the counter n. Preemptive multithreading systems provide mechanisms that effectively prevent thread switching at specific places in the code.

# Further reading & listening

1. APCS : The ARM Procedure Calling Standard

<http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042f/IHI0042F_aapcs.pdf>

1. Cortex M0 technical reference manual

<http://infocenter.arm.com/help/topic/com.arm.doc.ddi0432c/DDI0432C_cortex_m0_r0p0_trm.pdf>

1. Cortex M0 instruction set summary

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0432c/CHDCICDF.html>

1. Microchip SAM3X / SAM3A Series datasheet

<http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-11057-32-bit-Cortex-M3-Microcontroller-SAM3X-SAM3A_Datasheet.pdf>

1. A History of the ARM Microprocessor and it's Rise and Fall

<https://www.youtube.com/watch?v=_6sh097Dk5k>

1. ARM inventor: Sophie Wilson

<https://www.youtube.com/watch?v=jhwwrSaHdh8>

<https://www.youtube.com/watch?v=re5xAqgKqc0>

1. 2016-2017 note: the use of the heap will be covered in the V2CPSE2 course. [↑](#footnote-ref-1)
2. Note that even though it uses (mainly) 16-bit instructions, a Cortex is still considered a 32-bit processor because its registers, ALU and data paths are still (at least) 32 bit. [↑](#footnote-ref-2)
3. The SAM3X8E chip in an Arduino Due is a Cortex-M3 CPU, so it has a superset of the M0 instructions. [↑](#footnote-ref-3)
4. X is the maximum number of bits in the value, for instance immed8 can be a value in the range 0..255. [↑](#footnote-ref-4)