3

ARM Architecture

WHAT'S IN THIS CHAPTER?

- Understanding the basic terms
- Understanding a processor
- ARM processor internals
- Understanding program flow and interruption
- The different technologies

It doesn't matter if you are talking about an ARM processor, a 68k or even an x86 processor; they all have some common subsystems. There are slight differences in the way some subsystems are accessed, or the amount of subsystems present, but all processors retain the same basic principle, no matter what architecture. Following is a brief explanation of the core technology found in all modern processors, before going deeper into the specific details that make ARM processors what they are.

UNDERSTANDING THE BASICS

Everything in a computer is a number — text, images, sounds — everything is written down as a collection of numbers. A computer's job is to take some data and run operations on it; put simply, take some numbers, and do some mathematical computations.

The Colossus was the first programmable digital electronic computer, used in Britain during World War II to help in the cryptanalysis of the Lorenz cipher code, a cryptocode used by the German High Command to communicate with its armies. Colossus didn't actually completely decrypt messages; it was used to break Lorenz key settings using specialized code. Colossus could outperform humans both in terms of speed and reliability, and by the end of the war ten Colossus computers were in service.

While the Colossus was programmable, it was not a general purpose computer. It was designed with a specific task in mind and could not be programmed for another task, only programmed for different calculations on the same cipher. Programming was accomplished by setting up plugs and switches before reading in a cipher.

In America, the ENIAC was announced in 1946 and was used mainly to calculate ballistic trajectory, but was also used as a calculator for the hydrogen bomb project. What made it unique was its capacity of branching, or executing computer code depending on a previous result. Instead of a calculator simply adding numbers together, a computer could conditionally execute; take this number and multiply it by 2. Is the result less than 0? If it isn't, then subtract 20; otherwise, add 10. For ballistics, a computer could be told to continue calculating the speed, distance, and height of the projectile and to continue as long as the height is above sea level.

To function, a computer needs several things:

- A processor, on which all the work will be done
- Memory, to store information
- Input and output, to get information and to return it to the user (or activate outputs depending on certain conditions)

The processor is the ARM core. It might be an ARM Classic processor, such as an ARM11, or a Cortex, for example a Cortex-A8.

The memory might be more complicated. There are often several types of memory on an embedded system. There might be large amounts of DDR storage available, but DDR requires an initialization sequence, and therefore, memory. ARM systems often come with a small amount of internal memory (enough to start the system and to initialize any external systems) such as DDR memory, or possibly flash memory to read the operating system from.

Input and output can be almost anything and are either directly on the processor or SoC, or specific components mapped to external addresses.

Register

A CPU register is a small amount of fast memory that is built directly into the CPU and is used to manipulate data. ARM CPUs are load/store architecture, meaning that all calculation done on a CPU is done directly onto registers. First, the CPU reads from main memory into a register before making calculations, and possibly writing the value out into main memory. No instructions operate directly onto values in main memory. This might sound inefficient, but in practice it isn't; it saves having to write to main memory after each operation. It also significantly simplifies the pipeline architecture, something that is crucial for RISC processors.

At first glance, it might be surprising that there are so few registers on a system, but with a bit of careful work, most routines can be created using few registers. ARM processors actually have more registers than some.

Stack

The *stack* is a memory location in which temporary data is put and retrieved when needed. It is a LIFO: Last In, First Out. Some card games use a stack; a place in which cards can be put and retrieved but only in a specific order; the last card placed is the first one out. To get to a specific card, you must first remove all the other cards above it. The stack works in the same way.

The stack is primarily used when executing subroutines. When entering a subroutine, care must be taken to ensure that some registers retain their initial value, and to do that, their contents are placed onto the stack. So long as care is taken to take back the same amount of elements as was put into the stack, then the same values will always be read back.

The stack can fill up quickly, depending on the situation. During complicated calculations, variables must be pushed onto the stack to make room for new data. When calling a subroutine with complex arguments, they are often pushed onto the stack, and in the case of an object, the entire object is pushed onto the stack. This can result in huge increases in the stack size, so care must be taken to ensure that the stack does not overflow.

Internal RAM

Not all processors have internal RAM, but most do. It is often small compared to system memory, but it serves its purpose. On a typical system, there might be as much as 512 megabytes of external DDR memory, but DDR memory takes time to initialize. You need to do lots of steps to get DDR memory up and running, and in the meantime you cannot do your job with registers alone. Therefore, most ARM processors have a minimal amount of internal RAM, where you can transfer a program and run it, therefore setting up critical systems before switching to DDR memory. Internal RAM is also usually much faster than external RAM.

Cache

Early CPUs read instructions directly from the system memory, but when considering the time it takes to read data from the system memory compared to a processor cycle time, it was clear that a large portion of the CPUs' time was spent waiting for data to arrive. Writing data to the system memory was often even worse. Something had to be done, and so cache memory was developed.

CPU technology advancements mean that the speed of CPUs has grown many times in comparison to the access speed of main memory. If every instruction on a CPU required a memory access, the maximum speed of a CPU would be the maximum speed of the system memory. This problem is known as the *memory bottleneck*.

Cache memory is made from a special sort of memory, SRAM. SRAM, or Static RAM, has a speed advantage over DRAM, or Dynamic RAM. Unlike DRAM, SRAM does not need to be refreshed and can hold data indefinitely provided that it remains powered. SRAM provides high speed, but the cost is prohibitively high, so only a small amount of SRAM is available; main system memory is rarely SRAM.

Cache memory is used to store information recently accessed by a processor. Several layers of cache may be implemented: Cache (Level 1) is the closest to the CPU and the fastest. It is often relatively

small, varying between 4 KB and 64 KB in size. L2 cache (Level 2) is often slightly slower than L1 but also much larger. L2 cache can be in the range of 128 KB all the way to 4 MB or more.

There are two distinct cache architectures: Von Neumann (or unified) and Harvard. *Unified cache* is a single memory cache used for all memory zones. *Harvard cache* separates instruction cache and data cache. The separate caches are often referred to as D-cache (data cache) and I-cache (instruction cache). Harvard architectures have physically separate signals for storage for code and data memory, and Von Neumann architectures have shared signals and memory for code and data.

On boot, caches are disabled but not necessarily invalidated; they must be specifically set up and configured to function.

Read Cache Strategy

Cache memory is designed to avoid lengthy reads and writes, by reading in sections of memory into cache upon first use in the hope that future reads can then read from cache. If the processor requests some data, it first checks the cache. If it finds the data available, it is called a *cache hit*, and the data will be available immediately without having to read data from external RAM. If the data is not available, it is a *cache miss*, and the relevant data must be read from the system memory into the cache. Instead of reading in a single value, a cache line is read in.

Although this can be useful for some portions of the system memory, there are locations in which you do not want cache to operate at all. Caching the serial port could be disastrous; instead of reading from the serial register, you would constantly be reading the cache and presuming that no data is available. That is where memory management comes in — you can program specific memory zones to be cacheable or non-cacheable.

Memory management is done by the Memory Management Unit (MMU) or, for some systems, the Memory Protection Unit (MPU). For some Cortex-M systems, neither MMU nor MPU is available, and the cacheable attributes or memory regions are part of the fixed architectural map.

Of course, using this strategy, you are soon presented with a problem. When a cache-miss is encountered and the cache is full, what happens? In this case, one of the cache entries has to be evicted, leaving place for a new entry. But which one? This is one of the subjects on optimization; the trick is to know which cache entries will be used in the future. Because it is extremely difficult to know what will be used and what won't, one of the cache eviction strategies is LRU, or Least-Recently Used. By using this technique, the most recently accessed cache will remain, and older entries will be deleted. Careful planning can help optimize systems by defining which sections of memory are cacheable and which ones aren't.

Write Cache Strategy

Write-cache strategy is comparable to read-cache strategy, but there is a difference. Writing to cache can effectively speed up operation, but sooner or later the external memory needs to be updated, and you must first think about how to do that. There are two possible policies: write-through and write-back.

When writing data with the write-through policy, data is written to the cache, and at the same time written to system memory. Subsequent reads will read from the cache. Write-back cache is

slightly different from write-through. Initially, writing is done only to the cache; writing to external memory is postponed until the cache blocks are to be replaced or updated with new content. Cache lines that have been modified are marked as *dirty*. Write-back does have speed advantages but is more complex to implement and also has a drawback. A read miss in a write-back cache (requiring a block to be replaced by a data read) often requires two memory accesses: one to write the dirty cache line to system data and one to read system data into a new cache line.

GETTING TO KNOW THE DIFFERENT ARM SUBSYSTEMS

After having shown the basics, I will now show some of the subsystems on ARM processors. They are the components that make up an ARM processor, and are essential components to an ARM core.

Presenting the Processor Registers

An ARM core can be thought of as having 16 32-bit general registers; named `r0` to `r15`. In reality, however, there are several more because some registers are mode-specific. They are known as *banked registers*. Registers `r0` to `r7` are the same across all CPU modes; they are never banked. Registers `r8` to `r12` are the same across all CPU modes, except for FIQ. `r13, r14,` and `r15` are unique to each mode and do not need to be saved.

As you can see in Figure 3-1, when switching from User Mode to Fast Interrupt Mode, you still have the same registers `r0` to `r7`. That means that the values that were in `r0` to `r7` are still there, and when returning from Fast Interrupt, you return to where you were before the interrupt, and that portion of code expects to find the same values. However, `r8` to `r14` are "banked," meaning that these registers are used only inside your current mode of operation. The original `r8` to `r14` are still there and will be visible after you exit the fast interrupt. The advantage of this is speed; on returning, the registers must be set to their original values.

In normal programming, the user can freely access and write registers `r0` to `r12`. `r13`, `r14,` and `r15` are reserved for special purposes. The ARM coding conventions (the AAPCS, Procedure Call Standard for the ARM Architecture) state that when calling a subroutine, the arguments are passed in the first four registers (`r0` to `r3`), and return values are also passed in `r0` to `r3`. A subroutine must preserve the contents of the registers `r4-r11`. It is up to the subroutine to see if it necessary to push registers to the stack, or if it is possible to make required calculations on the first four registers. Whatever the decision, on returning, the caller function must have the result in `r1`, and the contents of `r4` to `r12` must be preserved.

r0 to r3

Normally, the first four registers, `r0` to `r3`, are used to pass arguments to a function. After these four registers have been used, any further arguments must be placed onto the stack. This can be configured with a compiler option. `r0` is also used as the return value of a function. If the return value is more than 32-bits wide, `r1` is also used. A program must assume that any function call will corrupt `r0` to `r3`.

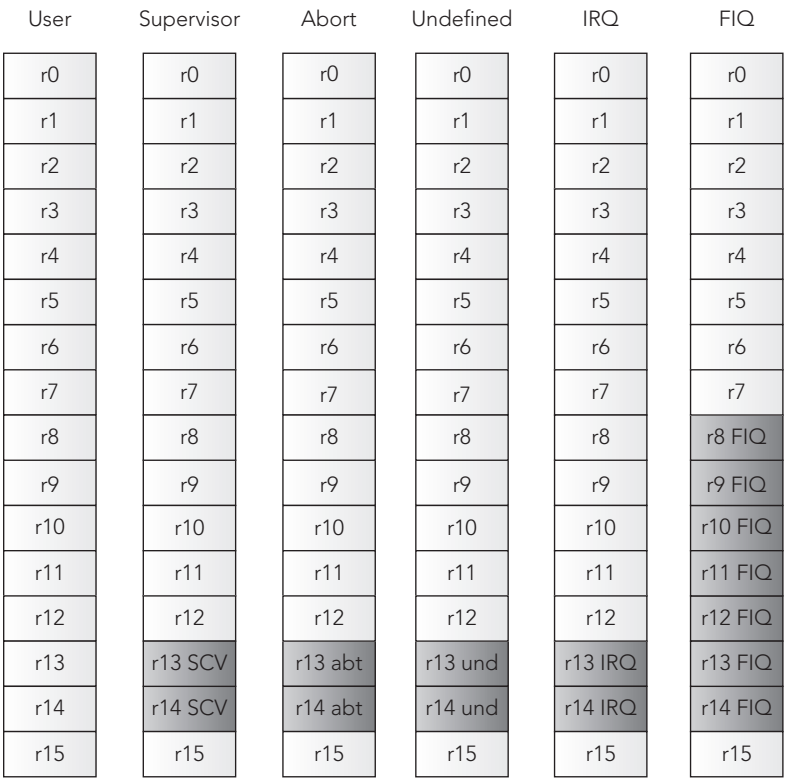

FIGURE 3-1: ARM Registers in different modes

r4 to r11

`r4` to `r11` are general purpose registers and can be used for any calculation, but they must be preserved by a function if their values are changed.

r12

`r12` is sometimes known as the IP register and can be used as an interprocess scratch register. The exact use depends heavily on the system being used, and in some cases, it is used as a general purpose register. If you use an operating system, refer to the operating system guide as to the usage of `r12`. If you are creating a bare metal system, you can use `r12` as you see fit.

The AAPCS states that `r12` may be corrupted by any function call, so programs must assume that it will not be preserved across a call.

r13: The Stack Pointer

The stack pointer is an important register. `r13` has a special function; it is the stack pointer. Just like the other registers, it is possible to read and write to this register, but most dedicated instructions will change the stack pointer as required. It is necessary to set up this register by writing the correct

address, but after that, it is no longer necessary to directly change this register. Thumb even forbids changing the stack pointer, with the exception of add and subtract.

When entering a function, `r4` to `r11` need to be returned to their initial values before leaving. To do that, use the PUSH and POP instructions, both of which modify the SP as required. Of course, it is not efficient to automatically PUSH and POP all the registers; therefore, the compiler will look and see what is to be done and will operate only on the registers that need to be saved, for example:

```
subroutine   PUSH {r0-r3,r12,lr} ; Push working registers and the link register
             BL my_function
             ; my_function will return here
             POP {r0-r3,r12,pc} ; Pop working registers, r12 and the PC
```

r14: The Link Register

`r14` holds the value of the Link Register, the memory address of an instruction to be run when a subroutine has been completed. Effectively, it contains the memory address to return to after you finish your task. When the processor encounters a branch with link instruction, a `BL`, `r14` is loaded with the address of the next instruction. When the routine finishes, executing `BX` returns to where the program was.

Here is an example:

```
AREA    subrout, CODE, READONLY   ; Name this block of code
ENTRY                 ; Mark first instruction to execute
start   MOV     r0, #10           ; Set up parameters
        MOV     r1, #3
        BL      doadd             ; Call subroutine

[ ... ]

doadd   ADD r0, r0, r1 ; Subroutine code
        BX lr; Return from subroutine

        END ; Mark end of file
```

r15: The Program Counter

`r15` holds the value of the Program Counter, the memory address of the next instruction to be fetched from memory. It is a read/write register; it can be written to, as is sometimes the case when returning from a branch instruction, modifying the address of the next instruction to be executed.

There is, however, a trick. Although technically the PC holds the address of the next instruction to be loaded, in reality it holds the location of the next instruction to be loaded into the pipeline, which is the address of the currently executing instruction plus two instructions. In ARM state, this is 8 bytes ahead, and in Thumb state it is 4 bytes. Most debuggers will hide this from you and show you the PC value as the address of the currently executing instruction, but some don't. If, during your debugging session, the PC points to something that doesn't seem related, check the documentation to see what the PC is supposed to show.

Presenting the CPSR

The CPSR is technically a register but not like the registers `r0` to `r15`. The CPSR, short for Current Program Status Register, is a critical register that holds the status of the running program and

is updated continuously. It contains condition code flags, which may be updated when an ALU operation occurs. Compare instructions automatically update the CPSR. Most other instructions do not automatically update the CPSR but can be forced to by adding the `S` directive after the instruction.

The ARM core uses the CPSR to monitor and control internal operations. The CPSR holds the following, among others:

- Current processor mode
- Interrupt disable flags
- Current processor state (ARM, Thumb, Jazelle, and so on)
- Data memory endianness (for ARMv6 and later)
- Condition flags

CPSR specifications may vary slightly from one architecture to another as ARM implements new features.

If the CPSR is the Current PSR, the SPSR is the Saved PSR. When an ARM processor responds to an event that generates an exception, the CPSR is saved into the SPSR. Each mode can have its own CPSR, and when the exception has been handled, the SPSR is restored into the CPSR, and program execution can continue. This also has the advantage of returning the processor to its exact previous state.

Understanding Condition Flags

The ALU is connected directly to the CPSR and can update the CPSR registers directly depending on the result of a calculation (or comparison).

N – Negative

This bit is set if the result of a data processing instruction was negative.

Z – Zero

This bit is set if the result was zero.

C – Carry

This bit is set if the result of an operation was greater than 32 bits.

V – Overflow

This bit is set if the result of an operation was greater than 31 bits, indicating possible corruption of the signed bit in signed numbers.

In 2's complement notation, the largest signed 32-bit number a register can hold is `0x7fffffff`, so if you add `0x7fffffff` and `0x7fffffff`, you can generate an overflow because the result is larger than a signed 32-bit number, but the Carry (C) is not set because you do not overflow an unsigned 32-bit number.

Interrupt Masks

Interrupt masks are used to stop (or allow) specific interrupt requests from interrupting the processor. It is often useful to disable interrupts for specific tasks before re-enabling them. When servicing an IRQ, further IRQs are disabled and FIQs are not modified. When servicing a fast interrupt, FIQs and IRQs are disabled; that way, critical code cannot be interrupted. When the interrupt operation is over, the SPSR is restored and the processor is returned to its previous state (with the previous settings for interrupts).

On some cores, there is a special interrupt that cannot be disabled — NMI, or Non-maskable Interrupt.

Calculation Unit

The calculation unit, as shown in Figure 3-2, is the heart of an ARM processor.

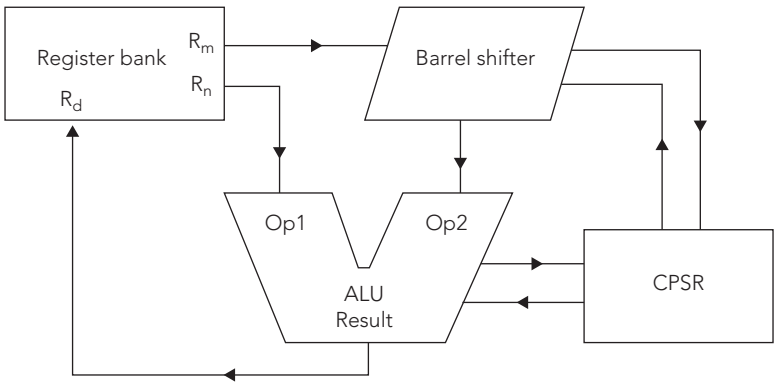

FIGURE 3-2: ARM Calculation Unit

The Arithmetic Logic Unit (ALU) has two 32-bit inputs. The first comes directly from the register bank, and the second one comes from the shifter. The ALU is connected to the CPSR and can shape the calculation output depending on the CPSR contents and also update CPSR contents according to the results of a calculation. For example, if a mathematical operation overflows, the ALU can update the CPSR directly.

You must understand that ARM cores in ARM mode do not actually need shift instructions, contrary to many other processors. Instead, the barrel shifter can perform a shift during an instruction by specifying a shift on the second operator directly inside an instruction. In Thumb mode, the instructions are simplified and shift instructions do exist.

Pipeline

The pipeline is a technique used in the design of ARM processors (and others) to increase instruction throughput. Instead of having to fetch an instruction, decode the instruction, and then execute it, you can do all three at the same time but not on the same instruction.

Imagine a factory. Imagine a worker inside the factory making a family computer. He first gets the mainboard and puts it inside the chassis. Then he takes the processor and puts it on the mainboard. Then he installs the RAM, and finally, he installs the graphics card. One worker has made the entire computer; one person is responsible for the entire chain. That isn't how they are made, though; computer manufacturers rely on assembly lines. One person will put the mainboard inside the chassis, and instead of continuing to the next task, he will repeat the process. The next worker on the assembly line will take the work of the first person and install the processor onto the mainboard, and again, the next task will be performed by someone else. The advantage is that each task is simplified, and the process can be greatly accelerated. Since each task is simple it can be easily duplicated, making several lines, parallelizing fabrication, and doubling the output.

Although making a desktop PC might seem relatively simple, imagine the complexity of a laptop computer, a flat-screen TV, or a car. Each task, although simple compared to the entire product, is still complex.

A CPU pipeline works in the same way. For example, one part of the processor constantly fetches the next instruction, another part "decodes" the instruction that has been fetched, and finally, another part executes that instruction. CPUs are driven by a clock; by doing more things on each clock pulse, you can increase the throughput of the CPU, and since each operation is made simpler, it becomes easier to increase the clock speed, further increasing throughput.

The advantage is, or course, speed. However, there are disadvantages of a pipelined system, notably stalls. A stall occurs when a pipeline cannot continue doing work as normal. For example, Figure 3-3 shows a typical six-stage pipeline.

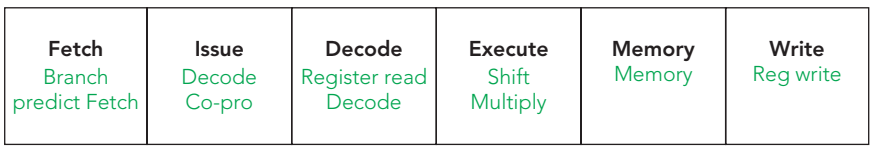

FIGURE 3-3: ARM six stage pipeline

Stage 3 accesses any operands that may be required from the register bank. After the calculation is done, in stage 6, you can write the results back into the register bank. Now suppose you have this:

```
MOV r5, #20
MOV r8, [r9]
ADD r0, r1, r2
SUB r0, r0, r3
MOV r4, r6
MVN r7, r10
```

Each instruction is run sequentially. The first Move instruction moves the value 20 into `r5`. The second Move instruction requires a fetch from memory, and assuming that this data is not located in cache, it can take some time. However, the data will be written into `r8`, and the instructions behind it do not require `r8`, so they would be stalled, waiting for an instruction to finish without even requiring the result.

There are different techniques to avoid stalls. One of the main reasons for stalls are branches. When a branch occurs, the pipeline needs to be filled with new instructions, which are probably in a different memory location. Therefore, the processor needs to fetch a new memory location, place the first instruction at the beginning of the pipeline, and then begin working on the instruction. In the meantime, the "execution" phase has to wait for instructions to arrive. To avoid this, some ARM processors have branch prediction hardware, effectively "guessing" the outcome of a conditional jump. The branch predictor then fills in the pipeline with the predicted outcome. If it is correct, a stall is avoided because instructions are already present in the pipeline. Some branch predictors have been reported to be 95 percent correct. More recent branch predictors even manage a 100 percent mark by speculatively fetching both possible execution paths, and discarding one of them once the outcome of the branch is known.

There are several cases where the order of instructions can cause stalls. In the previous example, the result of a memory fetch wasn't required, but what would happen if the instruction immediately afterwards required that result? Pipeline optimization would not be able to counter the stall, and the pipeline might stall for a significant amount of time. The answer to this is "instruction scheduling," which rearranges the order of instructions to avoid stalls. If a memory fetch might stall a pipeline, a compiler may place the instruction earlier, thus giving the pipeline a little more time.

Another technique used on some processors is known as out-of-order execution. Instead of the compiler rearranging instructions, the ARM core can sometimes rearrange instructions itself.

Tightly Coupled Memory

Cache can greatly increase speed, but it also adds problems. Sometimes, you need data to be stored in memory that isn't cacheable to be certain of the contents. Sometimes, you also want data to be "always available" and to have the speed of something in cache but without using up all the system cache. When reading data, if you have a cache hit, the data is immediately available, but if you have a cache miss, that data must be read from the system memory, often slowing down the system, in some cases considerably. You want critical interrupt handler code to be "always available," interrupt stacks, or mathematical data if the calculations require vast amounts of raw data.

Tightly Coupled Memory (TCM) is available on some processors. When available, TCM exists in parallel to the L1 caches and is extremely fast (typically one or two cycles' access time). The TCM is consequently not cacheable, leaving the cache free for other instructions and data.

TCM is like internal RAM, only configurable. By setting registers in the CP15, you can select separate instruction-side and data-side memory, either instruction or data-side memory, or complete deactivation of the TCM. It can be placed anywhere in the address map, as long as it is suitably aligned.

Coprocessors

ARM processors have an elegant way to extend their instruction set. ARM processors have support for coprocessors; secondary units that can handle instructions while the processor continues doing work.

The coprocessor space is divided into 16 coprocessors, numbered 0 to 15. Coprocessor 15 (CP15) is reserved for control functions, used mainly for managing caches and configuring the MMU. CP14 is reserved for debug, and CP 10 and 11 are reserved for NEON and VFP.

For classic processors, when a processor encounters an unknown instruction, it offers that instruction to any coprocessor present. Each coprocessor decodes the instruction to see if it can handle the instruction and signals back to the processor. If a coprocessor accepts the instruction, it takes the instruction and executes it using its own registers. If no coprocessor can handle the instruction, the processor initiates an undefined instruction exception. This is an elegant solution because some software enables "soft" coprocessors. If a coprocessor is not present, the instruction is caught during an exception and executed in software. Although the result is naturally slower than if the coprocessor was present, it does mean that the same code can be run, regardless of the availability of a specific coprocessor.

This system no longer exists; Cortex processors do not have a coprocessor interface, and instead, the instructions have been implemented into the core pipeline. Coprocessor instructions still exist and documents will still talk about the CP15 or other coprocessors; however, in order to simplify the core, the older coprocessor structure has been removed, but the instructions became valid ARM instructions. The coprocessor interface bus has 224 signals in it, so simplifying the coprocessor design was an important step to making processors simpler and faster.

CP15: The System Coprocessor

CP15 is a coprocessor interface developed by ARM and present on almost all processors except for the Cortex-M range.

The CP15's role is to handle system configuration: data cache, tightly coupled memory, MMU/MPU, and system performance monitoring. They are configured using the MRC/MCR instructions and can be accessed only in privileged modes. The registers are processor-specific; refer to your manual for more detailed information.

CP14: The Debug Coprocessor

The CP14 provides status information about the state of the debug system, configuration of certain aspects of the debug system, vector catching, and breakpoint and watchpoint configuration.

UNDERSTANDING THE DIFFERENT CONCEPTS

Before using an ARM processor, you need to know a few concepts. These concepts are the basis of ARM systems; some are related to the embedded systems world; others are purely ARM.

What Is an Exception?

Microprocessors can respond to an asynchronous event with a context switch. Typically, an external hardware device activates a specific input line. A serial driver might create an interrupt to tell the CPU that data is ready to be read in, or maybe a timer that sends signals periodically. It is the hardware's way of saying, "I have something I need done." This makes the processor do something that is called a *context switch*; the processor stops what it was doing and responds to the interrupt. Imagine working at your desk, when the phone rings. This forces you into a context switch. You make a mental note of what you were doing, you acknowledge that the phone is ringing, and now you are free to choose what to do next. You could answer it, and what you were doing before has to wait. You could send the call to someone else or even ignore the call. Whatever you choose,

you return to your previous task where you left off. For a processor, it is the same thing. When an interrupt arrives, you initiate a context switch. Registers can change, the current status is updated, and the current memory address is saved so that you can return later.

During its life cycle, a processor runs a program. Anything that disturbs that operation is called an *exception*. A software or hardware interrupt, a data abort, and an illegal instruction change the normal execution of a processor and are all exceptions. Even a reset is called an exception. When an exception occurs, the PC is placed onto the vector table at the corresponding entry, ready to execute a series of instructions before returning to what the processor was doing before (except for the reset exception). Several exceptions are available with different priorities.

Reset

A Reset exception has the highest priority because this is an external action that will put the processor in Reset state. When a CPU is powered on, it is considered to be in a Reset state. From here, you probably need to initialize all the hardware. When starting in Reset state, the core is in Supervisor mode, with all interrupts disabled.

Data Abort

A Data Abort happens when a data memory read or write fails. This can be for several reasons, but mostly it occurs when reading from or writing to an invalid address. When a Data Abort happens, the CPU is put into Abort mode, IRQ is disabled, FIQ remains unchanged, and `r14` contains the address of the aborted instruction, plus 8.

IRQ Interrupt

An IRQ interrupt occurs when an external peripheral sets the IRQ pin. It is used for peripherals to indicate that they are awaiting service and need the CPU to do something. Some examples are an input device indicating that the user has entered data, a network controller indicating that data has arrived, or possibly a communication device indicating that it is awaiting data. Frequently, IRQs are also used by a timer, periodically sending an interrupt every few milliseconds, or microseconds. This is known as a *tick*.

FIQ Interrupt

An FIQ is a special type of interrupt designed to be extremely fast. It is mainly for real-time routines that need to be handled quickly. It has a higher priority than an IRQ. When entering FIQ mode, the processor disables IRQ and FIQ, effectively making the code uninterruptable (except by a data abort or reset event) until you manually reactivate the interrupts. These are designed to be fast, very fast, meaning that they are normally coded directly in assembly language. Also, FIQ is located at the end of the vector table, so it is possible (and common) to start the routine right there, instead of branching, saving a few instructions.

Prefetch Abort

The Prefetch Abort exception occurs when the processor attempts to execute code at an invalid memory address. This could happen for several reasons: The memory location might be protected and memory management has specifically denied access to this memory, or maybe the memory itself is not mapped (if no peripherals are available at that address).

SVC

A Supervisor Call (SVC) is a special software instruction that generates an exception. It is often used by programs running inside an operating system when requesting access to protected data. A non-privileged application can request a privileged operation or access to specific system resources. An SVC has a number embedded inside, and an SVC handler can get the number through one of two methods, depending on the core. Most processors embed the SVC number inside the instruction, and some Cortex-M processors will push the SVC number to the stack.

Undefined Instruction

An Undefined Instruction occurs when the ARM core reads an instruction from memory, and the recovered data does not correspond to an instruction that the ARM core can execute. Either the memory read does not contain instructions, or it is indeed an instruction that the ARM core cannot handle. Some Classic processors used this technique for floating point instructions; if the processor could execute the instruction, it would use hardware-accelerated routines, but if the processor did not support hardware floating point, an exception would occur and the processor would use software floating-point.

Handling Different Exceptions

Exceptions exist not only to warn the processor, but also to perform different actions. When handling an interrupt exception, you need to do some work before returning to the main application, but when handling a Data abort, you might think that all is lost. This isn't always the case, and the exception actually exists to avoid everything grinding to a halt. Every Linux developer has, sooner or later, been confronted with the dreaded Segmentation Fault. A *segfault* is, generally, an attempt to access a memory address that the program does not have the right to access, or memory that the CPU cannot physically access. The exception is "trapped"; the operating system takes control and stabilizes the system. This sometimes means that the offending program is terminated, but more often it is just the program's way of telling the operating system that it requires more resources. An application may overflow its stack, in which case the operating system can choose to allocate it some more, or if an application runs off the end of the current code page, the operating system will load and map the next page.

When the operating system finishes handling the exception, it returns control to the application, and the system keeps going.

When an exception occurs, the core copies the CPSR into `SPSR_<mode>` and then sets the appropriate CPSR bits. It then disables interrupt flags if this is appropriate, stores the "return address" into `LR_<mode>`, and then sets the PC to the vector address. Note that if the CPU is in Thumb state, it may be returned to ARM state. Most Classic processors could only handle exceptions in ARM state, but the ARM1156 and all Cortex processors can be configured to handle exceptions in ARM or in Thumb state.

To return from an exception, the exception handler must first restore the CPSR from `SPSR_<mode>` and then restore the PC from `LR_<mode>`.

Modes of Operation

An ARM core has up to eight modes of operation. Most applications run in User mode, and the application cannot change modes, other than by causing an exception to occur. The modes other than User mode are known as *privileged modes*. They have full access to system resources and can change modes freely. Five of them are known as *exception modes*; they are entered when specific exceptions occur. Each of them has some additional registers to avoid corrupting User mode state when the exception occurs. They are FIQ, IRQ, Supervisor, Abort, and Undefined mode. Some cores also have a further mode — Monitor mode — that enables debugging a system without stopping the core entirely.

User Mode

Normally a program runs in User mode. In this mode, the memory is protected (if the CPU has an MMU or an MPU). This is the standard mode for applications, and indeed, most applications can run entirely in User mode. The only way a program running in User mode can change modes directly is to initiate an SVC. External events (such as interrupts) can also change modes.

System Mode

System mode is a mode that can be entered only via an instruction that specifically writes to the mode bits of the CPSR. System mode uses the User mode registers and is used to run tasks that require privileged access to memory and coprocessors, without limitation on which exceptions can occur during the task. It is often used for handling nested exceptions, and also by operating systems to avoid problems with nested SVC calls.

Supervisor Mode

Supervisor mode is a privileged mode that is entered whenever the CPU is reset or when a `SVC` instruction is executed. Kernels will start in Supervisor mode, configuring devices that require a privileged state, before running applications that do not require privileges. Some bare metal systems run almost entirely in Supervisor mode.

Abort Mode

Abort mode is a privileged mode that is entered whenever a Prefetch Abort or Data Abort exception occurs. This means that the processor could not access some memory for whatever reason.

Undefined Mode

Undefined mode is a privileged mode that is entered whenever an Undefined Instruction exception occurs. This normally happens when the ARM core is looking for instructions in the wrong place (corrupted PC), or if the memory itself is corrupted. It can also happen if the ARM core does not support a specific instruction, for example when executing a VFP instruction on a core where VFP was not available. The undefined instruction was trapped and then executed in software, therefore emulating VFP.

Undefined mode can also occur on coprocessor faults — the coprocessor is present but not enabled; it is configured for privileged access, but access is attempted in User mode; or it rejected an instruction.

IRQ Mode

IRQ mode is a privileged mode entered whenever the processor accepts an IRQ interrupt.

FIQ Mode

FIQ mode is a privileged mode entered whenever the processor handles an FIQ interrupt. In this mode, registers `r8` to `r12` are banked, meaning that they are available for use without having to save their contents. Upon returning to the previous mode, the banked registers are restored to their original state.

Having private registers reduces the need for register saving and minimizes the overhead of context switching.

Hyp Mode

Hyp mode is a hypervisor mode introduced in ARMv7-A for the Cortex-A15 processor (and later) for providing hardware virtualization support.

Monitor Mode

Monitor mode is a special mode used for debugging, but with the advantage of not stopping the core entirely. The major advantage is the possibility for other modes to be called — in monitor mode, the core can be interrogated by the debugger but still respond to critical interrupt routines.

Vector Table

A vector table is a part of reserved memory where the processor looks for information when it enters a specific mode. The classic model is used in pre-Cortex chips and current Cortex-A/R chips. In it, the memory at 0 contains several exception handlers. A typical vector table looks something like this:

```
00000000   LDR   PC, =Reset
00000004   LDR   PC, =Undef
00000008   LDR   PC, =SVC
0000000C   LDR   PC, =PrefAbort
00000010   LDR   PC, =DataAbort
00000014   NOP
00000018   LDR   PC, =IRQ
0000001C   LDR   PC, =FIQ
```

Upon entering an exception, the corresponding instruction is executed. Typically, in this part of the code, there will be jump instructions, with the possible exception of the FIQ exception. Because FIQ is at the end of the table, it is possible to put instructions here, avoiding the need for a jump and speeding up execution.

There is also an option called *high vectors*. Available on all ARM processors from ARM720T onwards, this option allows the vector table to be placed at `0xffff0000`, and can be configured by software control to relocate the table at any time.

The table is usually called a vector table, but that isn't always true. The vector table can contain one ARM instruction per entry, so they are generally jump instructions. However, it can also contain one 32-bit Thumb instruction, or two 16-bit Thumb instructions.

On Cortex-M chips, this is different. The vector table actually does contain vectors and not instructions. The first entries in a typical vector table on a Cortex-M chip might look something like this:

```
__Vectors     DCD     __initial_sp        ; Top of Stack
              DCD     Reset_Handler       ; Reset Handler
              DCD     NMI_Handler         ; NMI Handler
              DCD     HardFault_Handler   ; Hard Fault Handler
              DCD     MemManage_Handler   ; MPU Fault Handler
              DCD     BusFault_Handler    ; Bus Fault Handler
              DCD     UsageFault_Handler  ; Usage Fault Handler
```

This means that a Cortex-M can address the entire memory space, not just the memory space limited by branch commands.

Memory Management

Memory management is done through the Memory Management Unit (MMU), which enables you to control virtual-to-physical memory address mapping, enabling the processor to transparently access different parts of the system memory. An address generated by an ARM processor is called a *virtual address*. The MMU then maps this address to a physical address, enabling the processor to have access to the memory. The memory might be mapped "flat," in which the virtual address is equal to the physical address.

Another function of the MMU is to define and police memory access permissions. This control specifies whether a program has access to a specified memory zone, and also if that zone is read-only or read-write. When access to the memory zone is not permitted, a memory abort is performed by the processor. This can be essential for protecting code because a privileged application should not read system memory, and especially not modify it. It also enables several applications to run in the same virtual memory space. This requires a little more explanation.

The operating system allocates a certain amount of processor time for each running application. Before switching to the application, the operating system sets up the MMU for that particular application. The application happily runs until it is interrupted by the operating system, and during this time, it believes that it is running at a certain memory location. However, with the mapping of virtual addresses and physical addresses, a program might think it is running at, for example, `0x4000`, whereas the operating system has allocated it at `0x8000` in physical memory. When the operating system interrupts the application to enable some processor time to another application, it again reconfigures the MMU. Another program is then run, again thinking that it is running at `0x4000`, but the operating system might have allocated it at `0x9000`, and so on. It also means that one application cannot access the memory location of another application.

There are many uses for this that have been used throughout computer history. It can be used to relocate the system kernel and has been done on most large operating systems. It can also be used to access more than the physically available memory, by carefully switching memory and writing some memory to disk, flash, or some other form of mass storage.

ARM systems also have a distinct use for memory management. ARM CPUs boot from memory location `0x00000000` (or `0xffff0000` if high vectors are enabled), but this presents a problem. `0x00000000` must be located in ROM on first boot, but it is often useful, if not required, to change the vector table later, meaning that it must be located in RAM. In this case, MMUs can be used

to remap the memory; to place the boot ROM elsewhere in memory, and to map fast memory to the position of the vector table. This is illustrated in Figure 3-4.

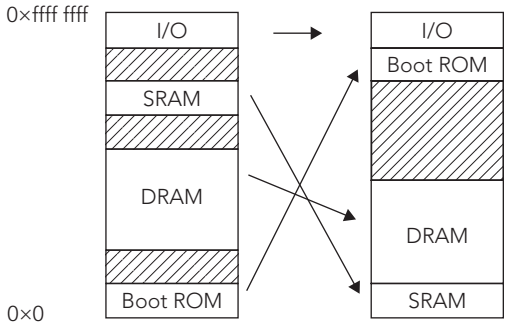

FIGURE 3-4: Memory remapping example

What Is Virtual Memory?

Every address generated by the processor is a virtual address. When the MMU is not enabled, either in reset state or because it was never configured, the memory is flat mapped between virtual and physical. When the MMU is configured and enabled, the processor requests are "translated" by the MMU without the processor actually knowing about the modification. The CPU might think that it is retrieving memory from the DDR2 chip at location `0x2080f080`, but the MMU might have changed the request to `0x9080f080`.

How Does the MMU Work?

The MMU needs information about the different translations, and to do that, it needs a set of translation tables. This is a zone in memory that contains information about the translations, separated into different sizes.

ARM MMUs support entries in the translation tables that can represent a 1 MB Section, a 64 KB Large page, a 4 KB Small page, or a 1 KB tiny page.

The first part of the table is known as the first-level table. It divides the full 4 GB address space into 4096 1 MB Sections, the largest size available. At the least, an MMU page table should contain these 4096 Section descriptors.

Each entry is called a first-level descriptor and can be one of four different items:

- A 1 MB Section translation entry, mapping a 1 MB region to a 1 MB physical region
- An entry to a second-level table for more precision
- Part of a 16 Mb Supersection
- A fault entry that's a 1 MB Section of unreadable data

Each of the 4096 entries describes memory access for a 1 MB Section. Of course, sometimes it is necessary to have a much smaller zone, for example, to have a 1 KB zone at the end of the stack to cause an exception if the stack grows too much, instead of potentially overwriting code or data. That is why a first-level descriptor can point to another memory location, containing a second-level table.

All this data is stored in system memory, but the MMU contains a small cache, the Translation Lookaside Buffer (TLB). When the MMU receives a memory request, it first looks in the TLB and resorts to reading in descriptors only from the tables in main memory if no match is found in the TLB. Reading from main memory often has an impact on performance, so fast access to some translations can become critical. A real-time system may need to access data in a specific region

quickly but not often. In the normal case, when a TLB entry is not used, it is replaced by another line that is used often. To react as quickly as possible, therefore, some TLB entries can be locked, meaning that they will always be present and never replaced.

PRESENTING DIFFERENT TECHNOLOGIES

The term *technology* refers to technological advances integrated as default over time. For example, when ARM introduced the Thumb technology, it was an option for the ARM7 processor (used in the ARM7TDMI). Thumb is now included by default in the ARMv5T architecture and all later versions.

JTAG Debug (D)

The Joint Test Action Group (JTAG) was an industry group formed in 1985 and whose aim was to develop a method to test circuit boards after manufacture. At that time, multilayer printed circuit boards were becoming the norm, and testing was extremely complicated because most pathways were not available to probes. JTAG promised to be a way to test a circuit board to detect faulty connections.

In 1990, Intel released the 80486, with integrated JTAG support, which led to the industry quickly adopting the technology. Although JTAG was originally designed just to test a card, new uses were studied, especially debugging. JTAG can access memory and is frequently used to flash firmware. Coupled with the EmbeddedICE debugging attribute, it provides a powerful interface.

Enhanced DSP (E)

As ARM-powered devices were used for more and more digital media applications, it was necessary to boost the ARM instruction set by adding DSP instructions, as well as SIMD instructions.

Digital signal processing (DSP) is the mathematical manipulation of information to modify or improve it in some way. The goal of DSP is usually to measure, filter, and/or compress/decompress real-world analog signals. For example, DSP is used for music players, not only converting a compressed digital file into analog music, but beforehand by converting an analog sound, recorded in a studio, into a digital format. Typical applications are audio compression, digital image processing, speech processing, or general digital communications. The use of SIMD instructions can lead to increased performance of up to 75 percent.

DSP can be done on just about any processor, but the routines are extremely repetitive and time-consuming. By adding specialized instructions, more calculations can be done with less processing power. Some of the early mp3 players used ARM7EJ-S, with enhanced DSP instructions. The DSP instructions enabled mp3 decoding at low speeds with little battery usage, ideal for mobile devices. ARM released a highly optimized mp3 software library, but enhanced DSP works for almost all digital information signals. The DIGIC processor is Canon's processor for its line of digital cameras. The Canon EOS 5D Mark III is a professional 22 megapixel camera, and by using a DIGIC 5, it can take 6 photos a second, apply noise reduction routines, save each image in raw output, and also convert it into JPEG, while keeping enough processor power available to keep the camera's functions running, notably the auto-focus based on the analysis of 63 points.

On Cortex-A class processors, this has been enhanced with NEON, optionally present on Cortex-A processors.

Vector Floating Point (F)

Vector Floating Point was introduced enabling hardware support for half-, single-, and double-precision floating points. It was called Vector Floating Point because it was developed primarily for vector operations used in motion control systems or automotive control applications.

Originally developed as VFPv1, it was rapidly replaced by VFPv2 for ARMv5TE, ARMv5TEJ, and ARMv6 architectures. VFPv3 is optionally available in ARMv7-A and ARMv7-R architectures, using either the ARM instruction set or Thumb and ThumbEE. A synthesizable version was made available, the VFP9-S, as a soft coprocessor for the ARM9E family.

EmbeddedICE (I)

EmbeddedICE is a powerful debug environment, and cores supporting the EmbeddedICE technology have a macrocell included inside the ARM core for enhanced debugging.

The EmbeddedICE macrocell contains two real-time watchpoint units that can halt the execution of instructions by the core. The watchpoint units can be programmed to break under certain conditions, when values match the address bus, the data bus, or various signals. The watchpoints units can also be programmed to activate on data access (watchpoint) or instruction fetches (breakpoint).

Jazelle (J)

The Jazelle DBX (Direct Bytecode eXecution) was a technique that enabled Java bytecode to be executed directly on an ARM processor. With the advances in processor technology, it was no longer required to have specific Java bytecode acceleration, and this technology is now deprecated.

The first implementation of this technology was in the ARMv5TEJ architecture, with the first processor being the ARM 926EJ-S. At the time, ARM processors dominated the mobile phone sector, as it still does today. Mobile phones were becoming more and more advanced, and users were demanding more and more features. New programs and games could be installed onto a mobile phone, enhancing the user experience. These applications were mainly written in Java ME, a special form of Java designed for embedded systems.

Long Multiply (M)

M variants of ARM cores contain extended multiplication hardware. This provides three enhancements over the previous methods:

- An 8-bit Booth's Algorithm was used, meaning that multiplications were carried out faster, a maximum of 5 cycles.
- The early termination method was improved, meaning that some multiplications could finish faster under specific conditions.
- 64-bit multiplication from two 32-bit operands was made possible by putting the result into a pair of registers.

This technology was made standard for ARM cores using architecture ARMv4 and above, and ARM9 introduced a faster 2-cycle multiplier.

Thumb (T)

Thumb is a second instruction set generated by re-encoding a subset of the ARM instruction set in 16-bit format. Because it is an extension of ARM, it was logical to call it Thumb.

Thumb introduced 16-bit codes, increasing code density. On some systems, the memory was 16-bits wide, so it made sense to use 16-bit instructions. Reducing instructions to 16 bit also meant making sacrifices, so only branch instructions can be conditional, and only registers `r0` to `r7` are available to most instructions. The first processor to include Thumb was the ARM7TDMI. This chip went on to power devices like the Apple iPod, the Nintendo Game Boy Advance, and most of Nokia's mobile phone range at the time. All ARM9 processors and above include Thumb by default.

Thumb-2 technology was introduced in the ARM1156T2-S core and extends the limited 16-bit instruction set by adding 32-bit instructions. Thumb with Thumb-2 is a "complete" instruction set in the sense that it is possible to access all machine features, including exception handling, without recourse to the ARM instruction set.

Synthesizable (S)

ARM licenses their IP, and it is normally delivered in a hard-macro format. Some are synthesizable cores and are delivered to clients in source code form. Synthesizable cores can be flashed onto an FPGA component, and users can add their peripherals to the ARM core before flashing and testing. This can be extremely useful for prototyping and to create small series of processors, since some manufacturers provide FPGA chips with an embedded ARM core and enough programmable logic to add a large range of peripherals. Changes can be tested, and when the logic development is complete, the FPGA chips can be flashed with a final configuration — a custom design without buying an ARM license.

Soft-cores enable greater flexibility but often come at the price of speed. Soft-cores are normally clocked at a slower speed than hard-core variants but have significant advantages.

Today, an ASIC may integrate an entire processor, RAM, ROM, and several peripherals all on one chip and are all user definable. ARM synthesizable cores allow companies to make optimized ARM cores based on the design goal of the company. The core can be optimized for power consumption, performance, cache size — almost all processor parameters can be customized. The flexibility provided by this solution can be seen in the huge ARM ecosystem, where a large number of products exist with different characteristics, but all based on the ARM core.

Synthesizable cores started with the ARM7TDMI-S and exist for some ARM9, ARM10, ARM11, and Cortex cores. Today, almost all ARM cores are delivered in synthesizable forms to ARM licensees.

TrustZone

TrustZone is a security technology implemented by ARM on the ARM1176JZ-S and now an integral part of every Cortex-A class processor.

TrustZone is an approach to security by creating a second environment protected from the main operating system. Trusted applications run in a Trusted Execution Environment and are isolated by hardware. Designed for mobile applications, TrustZone enables users to run unsafe code, while protecting the core functionality.

For example, mobile telephone manufacturers often require this sort of functionality. On a mobile phone, it is quite possible to have two operating systems that run simultaneously. One of them is the "main" operating system, the system that is visually present on the screen, and enables you to download and install programs from the Internet. This environment is not secure; it is possible to download malware despite best intentions. A second operating system, this time a real-time OS, is responsible for the hardware side of the telephone, the modem. Both systems are separated for several reasons: partly because of the operating system (for example, Android can't be compiled for every modem on the market); and secondly for security, the operating system must not have access to core systems. Mobile telephone manufacturers don't like it when you flash a new version of the operating system or have access to factory settings. This is one of the fields where this technology can be useful.

NEON

NEON is ARM's wide Single Instruction Multiple Data (SIMD) instruction set. Historically, computers have always been running one single task at a time. An application may require millions of calculations, and each calculation will be done one at a time. In normal circumstances, this works fine. When working on multimedia files, this proved to be slow and in need of some optimization. Now say, for example, you want to turn a graphics image into a black-and-white image. You might look at each pixel, take the red, green, and blue components, take the weighted average, and then write back the data. For a 320×256 pixel image, you would have to do this 81,920 times. That isn't too bad. When working on a Full HD image, you are working on a 1920×1080 pixel image, meaning 2 million calculations. This is beginning to become huge. A 22-megapixel camera will output files in the range of 5760×3840 — 22 megapixels, so 22 million calculations. Suddenly, this becomes painfully slow. By using NEON instructions, operations can be done on multiple values packed into registers in a single cycle, allowing for higher performance.

big.LITTLE

ARM big.LITTLE processing is ARM's answer to mobile devices' energy consumption. Today's always-on mobile devices are difficult to predict, sometimes requiring little processing power and sometimes requiring enormous amounts of power. Take the example of a tablet with an estimated 8 hours of battery life. On standby, even if the screen is off, the device is still running in the background, connecting to Wi-Fi every so often to fetch e-mails, allowing some programs to run in the background, and especially, running an alarm clock that will go off in 10 minutes to wake you up. And it does. The screen turns on, an alarm sounds, and the device says good morning the best it can. Up until now, you haven't done anything actually CPU-intensive; a low-powered CPU could do the job just fine. One hour later, you are on a flight on the way to your vacation destination, and to kill time, you start playing a game. Now things start to change. The CPU is being used intensively, and power consumption goes up. The processor adapts by increasing the clock rate, and you get the most out of your tablet. One hour later, you still aren't past the last level, but they start serving

drinks, so you put your tablet down. The operating system detects that you no longer need as much processing power as you did before, and it scales down the frequency, therefore using less power. The problem with this is simple: More powerful processors use more energy, no matter what the clock frequency is. Even when doing "nothing," there is still more silicon to power, and each clock cycle does more and more things, costing energy. Running your application on a Cortex-A7 will cost less energy than a Cortex A-15, meaning more battery life, but the Cortex-A15 is more powerful than a Cortex-A7, meaning better applications. The ideal solution would be to have a Cortex-A15 that has the battery life of a Cortex-A7, but that isn't possible. It is, however, possible to have a Cortex-A15 and a Cortex-A7 in the same chip. Enter ARM's big.LITTLE technology.

ARM's big.LITTLE works on the principle that both processors are architecturally identical. Because both processors have the same architecture, they can also have the same applications that can, if needed, switch from one processor to another. Processes switch between the two processors, depending on what is needed and depending on instructions issued by the kernel. When using background applications, applications are run on the low-powered CPU, and when the system is under load, the processes are run on the faster CPU. ARM estimates that by using this technology, substantial energy savings can be achieved. In ARM's tests, web browsing used 50 percent less power, and background tasks, such as mp3 playing, used 70 percent less energy. All this is achieved transparently for applications; software changes are made only in the operating system's kernel scheduler. Software doesn't even need to know if it is running on a big.LITTLE enabled processor.

SUMMARY

In this chapter, I have explained the different subsystems of an ARM processor, and provided an explanation of the different options available on select processors. I have presented how the processor starts, what the vector table is, and how it is used for exceptions, I have explained the different registers and which ones are reserved, before presenting some service registers, and finally, basic memory management.

In the next chapter, I will give a brief introduction to assembly language, with an explanation of what it is used for and how it is still essential to know assembly for embedded systems. I will also go through an example assembly program, but please don't run away! Assembly isn't that difficult, and it can even be fun.

4

ARM Assembly Language

WHAT'S IN THIS CHAPTER?

- Introduction to ARM assembly
- Use of assembly
- Understanding condition codes
- Understanding addressing modes
- Understanding your first ARM assembly program

Assembly language is the most basic programming language available for any processor. It is a collection of commands, or instructions, that the processor can execute. A program is a list of instructions in a specific order telling the computer what to do. Just like a calculator, you can tell a processor to take a number, to multiply it by 2, and to give you the answer. However, you need to supply some more information; you need to tell the processor where to get the number and what to do with it afterward.

INTRODUCTION TO ASSEMBLY LANGUAGE

A processor runs machine code, and machine code can be specific from one processor to another. Machine code written for a 6502 cannot, and will not, run on a 68000. Machine code is a list of numbers that make no sense whatsoever to (most) humans. To make the programmer's life a little more bearable, Assembly language was invented. Assembly language consists of words and numbers, and although it isn't as easy to understand as the English language, it is much easier to understand than reading numbers or punch cards.

Assembly language enables programmers to write computer programs, telling the processor exactly what it must do.

TALKING TO A COMPUTER

Talking to a computer isn't as easy as you would think. Hollywood has made a good job of making you think that computers are highly intelligent, but they aren't. A computer can follow instructions, no matter how badly written they are. To write good instructions, you need to know exactly how a computer works. You learned about the memory and about input and output, but now here's a little more about the processor and what it contains.

All processors use registers, internal memory used for specific reasons. ARM processors have 16 registers, named `r0` to `r15`. But what exactly is a register?

A register is, put simply, a memory location that can contain one number. Remember when you were at school, and you had a written test in front of you. "How much is 5 times 3?" Instinctively, today, you would write down 15. The habit over the years makes you forget about what actually goes on, but try to think of it through a child's perspective. This is an operation that he does not immediately know the answer to, so he takes it step by step. Take the first number, 5, and put it into your memory. Then take the next number, 3, and put that into your memory, too. Then do the operation. Now that you have the answer, write that down onto the paper. This is illustrated in Figure 4-1.

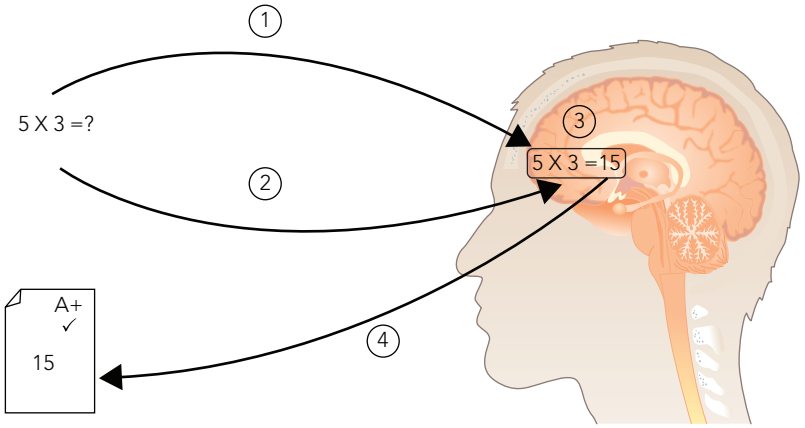

FIGURE 4-1: An example of mental calculation

A processor does the same thing. An ARM processor cannot do mathematical operations straight to and from memory, only registers. In this example, take the number 5 and load it into a register, let's say `r0`. Now, take the number 3, and load it into `r1`. Next, issue an instruction to multiply the value stored in `r0` by the value stored in `r1`, and put the result in `r2`. Finally, write the value stored in `r2` into the system memory. This is illustrated in Figure 4-2.

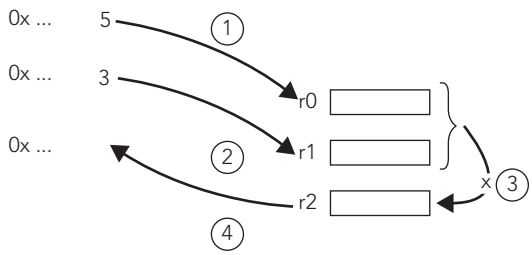

FIGURE 4-2: A calculation on an ARM processor

The question remains: Why do you need registers? Couldn't you do your operations directly into memory? The reason is simple, speed. By designing this functionality into the processor, it also makes the processor simpler, and reduces the amount of transistors required. This is one of the key factors in Reduced Instruction Set Computer processors.

WHY LEARN ASSEMBLY?

Assembly has been around since, quite literally, the beginning of processors. It is the lowest-level instruction set that a processor can use, and each processor has its own instruction set. Initially, everyone had to write computer programs in assembly. In today's world you have a choice of more than 100 programming languages, each with its strong points and weak points. Most low-level development today is done in C, a language that can be easily ported from one processor to another. It is easier to read than assembly and has many advantages over assembly. There is also another reason — portability. As seen previously, an assembly program written for one type of processor will not function for another type. Not all processors have the same instructions, or the same way of handling data. It is the C compiler's job to convert the C files into machine code for the correct processor. This might not seem important, since you may already know exactly what processor you will be using, but the C compiler knows about some of the optional features of the processor and can create optimized programs. Also, an external library might be designed to be used on a wide variety of processors, not just one specific processor.

So why would anyone need to learn assembly? Although languages such as C might present huge advantages, what you see is most certainly not what you get. It doesn't matter what language you choose — C, Python, Java, etc. — in the end, the only language a processor can use is assembly. When writing a program in C, the code is eventually compiled into assembly language. Although most programmers might not be concerned by assembly, embedded engineers will, sooner or later, be confronted by assembly code.

Embedded systems have two constraints that might not be as important for larger computer systems: speed, and size. Embedded systems often have to be as fast as possible and are usually heavily limited in terms of memory.

Speed

The Airbus A320 relies on a 68000 processor for the ELAC, the Elevator and Aileron Control. The 68000 was introduced in 1979, and although it is considered to be an "old" processor, it is also one of the most reliable. It is for this reason that it is used in mission-critical systems, but it comes at a price. It is not one of the fastest processors available, so all instructions must be carefully written and optimized to make sure that the chip runs as fast as possible.

This brings a question, one that sometimes surprises newcomers. Doesn't the compiler always create the most optimized code possible? The answer is no. They normally do a good job, but once in a while, they surprise you, or they won't quite understand exactly what it is you want to do. They can never be any better than you are. Imagine a shopping list. You have friends visiting, and you want to cook something for them, such as Chicken Basquaise. So, you start your list; you need a pound of tomatoes, a chicken (or six chicken breasts), four red peppers, three onions, some white wine, some thyme, and some basmati rice. And off you go with your recipe list. The list contains everything you

need. (Although you might add a few more ingredients here and there.) You have several choices as to what to do. You can get the ingredients as they appear on the list; start with the tomatoes, then get the chicken, and then go back to where the tomatoes were to get the peppers. Depending on the size of the supermarket, you can lose a lot of time. If you plan ahead, you could have at least grouped everything together. So this time, get the tomatoes, the red peppers, and the onions because they are in the same place. You don't need to backtrack to get something that was close to you. You have just optimized. The result is exactly the same, but it is quicker. But can you do anything else? Yes, you can; you can go even further. In the supermarket close by, there are two entries. Thinking cleverly about this, start with the tomatoes because they are close to the south entry. A few feet later, you can find the red peppers. From there, get the chicken. Going right, two lanes later, you find the white wine. Then continue your list getting the ingredients in the order that they appear while making your way to the north entry. Making the list would probably take much longer, but you now have the optimal list, one that takes you the shortest time possible. How much time would you spend on your shopping list? And how much time would you save with your new optimized path? Well, that depends. You won't spend an hour making a list if it saves only 8 minutes in the supermarket. If it can save 5 minutes, you will probably take 2 minutes to group the ingredients together. But what would happen if you had lots of friends and invited them over every weekend? Assuming that your friends wouldn't mind eating only Basquaise chicken, you could theoretically save 8 minutes each time, about 50 times a year? That 2-hour shopping list would have saved you a total of 6 hours in the supermarket.

Although this example is ridiculous, it serves a point. On an embedded system, there are parts of the program that you will run thousands, if not millions of times. A few milliseconds here and there could save you a lot of time later. It could also mean using a cheaper chip because you might not need the expensive 40MHz version since you were clever and managed to optimize everything so that the 20-MHz version would work.

Code from C and C++ can be compiled into machine language, and although the compilers normally do a good job, sometimes you have to write heavily optimized routines in assembly, or correct what the compiler is outputting. Also, some start-up routines cannot be written in C; to activate memory controllers or cache systems, you have to write some assembly code.

Rear Admiral Grace Hopper was one of the first programmers of the Harvard Mark I computer, an electro-mechanical computer built in 1944. She became obsessed with well-written code and often lectured on the subject. She became famous for her representation of a microsecond and a nanosecond, producing lengths of wire corresponding to the maximum distance that light could travel in that amount of time. When talking about a nanosecond, she produced a series of wires that were 11.8 inches long (29.97 cm). In comparison, she produced a wire corresponding to the maximum distance traveled by light in one microsecond, a total of 984 feet long (just under 300 meters). Producing a wire that long, she went on to say: "Here's a microsecond. Nine hundred and eighty-four feet. I sometimes think we ought to hang one over every programmer's desk, or around their neck, so they know what they're throwing away when they throw away a microsecond."

Size

A while ago, I was working on a bootloader for a mobile phone. A bootloader is the first piece of code that is run on an embedded system; its job was to test to see if a program existed on the

telephone. If such a program existed, there were cryptography checks to make sure that this was an official program. If the tests failed, or if no program was present, it put itself into a special mode, enabling a technician to download a new firmware via USB. To do that, we had to initialize the DDR memory, activate the different caches available on our CPU (in this case, an ARM926EJ-S), and activate the MMU. We also had to protect the bootloader; its job was to flash a new firmware but not give access to protected systems (the baseband, or confidential user information). We had to do all this in 16 Kb of NAND flash memories. Of course, in the beginning, there were huge ambitions; we could add a nice logo, a smooth menu interface, a diagnostics check, and so on. The list went on. When we released our initial binary, we were four times over the size limit. With a little optimization, we were three times over the limit. By getting rid of the fancy image, and the fancy menu, we were at 32 Kb. We started optimizing our C code, getting rid of a few functions here and there, and we came up with a binary just above 17 Kb. Several people tried to modify the C code, but we just couldn't get it below 16 Kb; we had to dig deeper, so we looked at the assembly code.

We soon realized that there were a few things that we could do, but we would shave off only a few bytes here and there. By changing how the program jumped to different functions, by modifying a few loops, and by repeating the process, we slowly made our way down to 16 Kb. In the end, we not only made the 16 Kb, but we also reduced the code further, allowing a few routines to be added.

Code compilers normally do a good job, but they aren't perfect. Sometimes they need a little bit of help from the developer.

Fun!

Writing in assembly can even be fun. No, seriously! In 1984, a new game called *Core War* was developed. It simulated the inside of a small computer. Two programs are inserted at random locations into the virtual memory, and in turn, each program executes one instruction. The aim of the game was to overwrite the other program and to control the machine.

In *Core War*, a dedicated language was used, called Redcode. However, it was rapidly "ported" to other systems, including ARM-based systems, as a fun way to learn programming. Battles were waged for entire evenings, with people testing their code. It wasn't that simple. Code couldn't be too large; otherwise, it might be damaged by the adversary. Different strategies were developed, and soon several "classes" became known, some often highly specialized in defeating another specific class. It was also an excellent way of teaching people what would happen if code was badly written.

Compilers Aren't Perfect

From time to time, you will be faced with a situation in which your product does not function as it should, but for some reason the code looks perfectly good.

On one occasion, I was confronted by a problem on an embedded system. We needed to read in a number from a sensor and to do a few calculations on that number to output a chart. There were three ways we could access the information, but one of them wouldn't work. We got a completely incoherent value, every time. The other two worked fine, so we knew that the sensor was working correctly, but these two ways were not available everywhere in our code. In one particular place, we had no choice but to use the third way. It didn't take us long to use a debugger, a Lauterbach Trace32. We were confident that we could find the problem immediately, using step by step, but

this just confused us more. The code was perfect, everything looked fine, but on one particular instruction, the output was meaningless. We had no choice but to dig deeper and look at the assembly code. It didn't take us long to realize that there was an alignment problem; instead of reading in one 32-bit integer from memory, the processor had to read in 4 bytes and do a quick calculation on all 4 to create a 32-bit integer, but the compiler failed to do this correctly, resulting in a corrupt value. Realigning the data fixed the problem immediately.

On another occasion, a basic routine didn't work as we wanted it to. We needed to loop 16 times and do a calculation each time. For some reason, we never managed to. The code was simple:

```
for (i = 0; i < 16; i++)
    DoAction();
```

Except it didn't work as intended. We checked: `i` was an integer, correctly defined. It worked before and afterward. Bringing up the assembly listing, we saw that it did indeed loop, but the variable was never initialized; the value of `i` was never set to 0. To this day, we use the same code, but just before, we set `i` to zero, and there is a comment explaining that the line must never be removed.

Understanding Computer Science through Assembly

Computers are a mystery to most people and embedded systems even more so. When talking to engineers, most understand the system, some understand what happens on lower layers, but today, relatively few understand what actually happens deep inside the mainboard. A good way to learn about what happens deep inside a CPU is to learn assembly. With Assembly language, you are at the lowest level possible and can understand what happens when you run a program.

Shouldn't You Just Write in Assembly?

Sooner or later, everyone asks, "Shouldn't I just write in Assembly? It's fast! It's lightweight!" Yes, but don't. There are very, very few projects that are written in assembly. Your time is valuable; *don't start in assembly*. Writing in C is much faster, and the compilers normally do an excellent job. If they don't, or if you need a routine that is highly optimized, carry on writing the code in C, and then look at what the compiler generates. You can save time doing this, and even if the end result is not 100 percent what you expect, the compiler probably does all the structuring that you want.

Writing in assembly does not automatically mean fast and elegant code, on the contrary. Just like with any language, it all depends on the quality of what you write; it is possible to make something in assembly that is slower than in C. Assembly is useful to know; you may face it several times in a single project, but with years of development, higher-level languages do make more sense.

Most experts agree; start by completing a project before looking to optimize it. Numerous tools exist to analyze code, and see what portions are called, and which portions take the most time to execute. It is always possible to return to some code and try to optimize, but working with highly optimized code from the start can be a nightmare. When you know where your CPU spends most of its time, then you can replace some parts with assembly.

USES OF ASSEMBLY

Few projects will be written entirely in assembly; using higher-level languages such as C just makes sense. They are quicker to develop and easier to maintain, and compilers do a good job in translating C to assembly. So, what exactly are the uses of assembly?

There are several reasons why assembly is still used, from bootloading the first steps of a project all the way to debugging a finished project.

Writing Bootloaders

You've almost certainly seen a lot of programs written in C, but the first instructions of a bootloader are generally written in assembly. Some routines, like setting up the vector tables, cache, and interrupt handling cannot easily be done in C. Also, some bootloaders need highly specialized code, either for size or speed, where assembly is needed.

Much of the processor's low-level configuration cannot be done in C; changing registers in a coprocessor requires assembly, and it cannot be done by writing memory. The first instructions of the Linux kernel are in assembly for this reason.

Reverse Engineering

Reverse engineering has been used from the beginning of the computer era, for good and for bad reasons. Sometimes it is necessary to see how a peripheral is initialized, and only the assembly code is available. Many drivers have been created this way, supporting devices made by companies that no longer exist, where no source code is available.

The Gaming Industry, Building a Better Mousetrap

As soon as the first computers became reasonably small, games have been available. People have always been fascinated with computer games, and today it is one of the biggest industries. The first medium for games was the good old cassette; an analog media that the new generation will probably never know. A standard tape player could be plugged into a computer to load programs directly. After a few minutes of watching colored bars on a screen, you were ready to play! And ever since the first games, software piracy has existed.

Copying tapes was ridiculously easy. High-end tape players could simply copy the audio from one tape to another, possibly degrading quality, but still allowing almost anyone to play a copy.

Game developers fought back. New systems were invented; questions were asked during the game. Upon reaching the doors to the city, a guard would ask, "What is the second word of the third paragraph on page 20 of your game manual?" Giving a wrong answer would mean never allowing you into the city, effectively stopping your game. Although it was possible to photocopy a manual, it did make things considerably more difficult for software pirates and also for people who actually did own the game.

Disk protection was also added. By directly modifying data on a disk's surface, a game could easily detect if the disk was an original. The disk copy program from the operating system would refuse to copy a sector that it thought to be in error, stopping disk copying. Again, systems were made that enabled disk copying, but it stopped most cases.

Hardware dongles were considered to be the "ultimate" protection. An application would look at the hardware on the computer, most often on the serial port, and ask a question. The dongle would provide an answer, and the program was effectively authenticated. Copying a dongle was very, very complicated. Most often, custom hardware chips were used, and the cost of creating a copy vastly outweighed the cost of a software license.

Software pirates changed their strategy. Instead of using hardware to make copies, they turned to software. Buried deep inside the application was a few lines of code of particular interest. Somewhere in the code, the application would look at something particular, receive an answer, compare that answer, and then either continue if the answer was correct, or stop if the answer was wrong. It didn't matter what language the program was initially written in; it all came down to assembly. By reading the assembly code, and finding out where the program looked for this particular piece of information, pirates could either force the answer to always be correct, or skip the question completely. Although this might sound easy, a program can be millions of lines of code long, and the portion that checks for copy protection might be as little as 10 lines. Also, the original code might have comments explaining what the developer was doing, or variables with useful and meaningful names, but not in assembly. There were certain techniques for helping pirates; serial ports mostly use the same address, so it was possible to analyze code looking for a specific address and then find out which of the results looked like the copy protection.

Software developers fought back. Copy protection was added into several parts of the code, making reverse engineering more difficult. Secondary routines checked to see if the primary routines hadn't been changed. False copy protection routines were added as a lure. Techniques became more and more sophisticated, but still someone came up with something to disable the copy protection features. Some do it for Internet fame, some do it to play the latest games, but some do it simply as a challenge.

Optimization

Most compilers do a good job at taking C files and converting them to assembly instructions. With a few command-line options, compilers can be told to optimize for speed or for size (or a mixture of both), but there are times when a compiler cannot correctly do its job and delivers functional code, but far from optimized.

ARM's weakness is division. More recent Cortex-A cores can do integer division, but previous cores had to do division in software — something that could take a lot of cycles to complete. When a function does one division, it isn't always necessary to optimize, but when a function does repeated calculations, sometimes several thousand times, it is often worthwhile to spend a little bit of extra time to see what can be done. Maybe a routine will divide only by 10, in which case a new function can be created, with tailor-made assembly instructions to get the job done as fast as possible.

ARM ASSEMBLY LANGUAGE

The ARM Assembly language is a well-designed language that, despite first impressions, can actually be easy to read. Where possible, it has been designed so that it can be easily read by a human, for example:

```
ADD r0, r1, r2
```

This instruction can look a little frightening at first, but it is easy. `ADD` is the shorthand for a mathematical addition. The three subsequent registers define the operation, but in what order? Well, a human would write the operation as `r0 = r1 + r2`, and that is exactly what is written here; `ADD` result = value 1 + value 2. The processor adds the value contained inside `r1` and the value contained inside `r2`, and puts the result into `r0`.

Layout

An assembly program source file is a text file and consists of a sequence of statements, one per line. Each statement has the following format:

```
label:     instruction     ;comment
```

Each of the components is optional.

- **Label** — A convenient way to refer to a memory location. The label can be used for branch instructions. The name can consist of alphanumeric characters, the underscore, and the dollar sign.
- **Comment** — All characters after an @ are considered comments and are there only to make the source code clearer.
- **Instruction** — Either an ARM instruction or an assembler directive

```
    .text
start:
    MOV r1, #20 ;Puts the value 20 into register r1
    MOV r2, #22 ;Puts the value 22 into register r2    ADD r0, r1, r2 ;Adds r1 and
        r2, r0 now contains 42
end:
    b end ;Infinite loop, always jump back to "end"
```

Instruction Format

This is the standard layout used in ARM assembly:

```
<op>{cond}{flags} Rd, Rn, Operand2
```

For example, the following code is used to add two registers together:

```
ADD R0, R1, R2
```

- `<op>` — Three-letter mnemonic, called the operand
- `{cond}` — Optional two-letter condition code
- `{flags}` — Optional additional flag
- `Rd` — Destination register
- `Rn` — First register
- `Operand2` — Second register or second operand

Condition Codes

You can add a two-letter condition code to the end of the mnemonic, allowing the instruction to be executed under certain conditions. For example, you can jump to some other code if the answer is equal to zero and continue otherwise. In the same way, you can branch to some new code if there is an overflow. This is mainly used when branching but can sometimes be used for other instructions. For example, you can tell the processor to load a register with a particular value if and only if a certain condition has been met. You see the command `MOV` later on, but put simply, `MOV` changes the value of a register. You can specify that you want the register to be changed, with a `MOV` command. However, you can also specify that you want the register to be changed if and only if the carry bit was set, with `MOVCS`, or if a previous compare was lower or the same, with `MOVLS`.

Condition codes look at the N, Z, C, and V flags on the CPSR (the CPSR is presented in Chapter 3). These flags can be updated with arithmetic and logical operations.

AL — Always

An instruction with this suffix is always executed. The majority of instructions are nonconditional; therefore `AL` is not required and may be omitted (and indeed should be omitted). For example, `ADD` and `ADDAL` are identical; they are both run unconditionally.

NV — Never

The opposite of `AL`, instructions with `NV` are never executed. Instructions with this condition are ignored. This code is now deprecated and shouldn't be used. It originally provided an analog for the `AL` condition code but was rarely used.

EQ — Equal

The instruction is executed if the result flag Z is set. If the Z flag is cleared, this instruction is ignored:

```
MOV r0, #42 ;Write the value 42 into the register r0
MOV r1, #41 ;Write the value 41 into the register r1
CMP r0, r1 ;Compare the registers r0 and r1, update CPSR register
BEQ label ;This command will not be run, since Z = 0
MOV r1, #42 ;Write the value 42 into the register r1
CMP r0, r1 ;Compare r0 and r1, update the CPSR
BEQ label ;This command will be run, since Z = 1
```

NE — Not Equal

The opposite of EQ, this instruction is executed if the Z flag is cleared. If the Z flag is set, this instruction is ignored:

```
MOV r0, #42 ;Write the value 42 into the register r0
MOV r1, #42 ;Write the value 42 into the register r1
CMP r0, r1 ;Compare the registers r0 and r1, update CPSR register
BNE label ;This command will not be run, since Z = 1
MOV r1, #41 ;Write the value 42 into the register r1
CMP r0, r1 ;Compare r0 and r1, update the CPSR
BNE label ;This command will be run, since Z = 0
```

VS — Overflow Set

This condition is true if the Overflow (V) bit is set, resulting in a mathematical operation that was bigger than the signed container (for example, adding together two 32-bit signed numbers that result in a 33-bit signed result).

VC — Overflow Clear

This condition is true if the Overflow (V) bit is clear. It is the opposite of VS and triggers only if the result of a mathematical operation was small enough to be held in its container. (For example, adding together two 32-bit signed numbers together resulted in a signed number that could be placed into a 32-bit signed container without data loss.)

MI — Minus

This condition is true if the Negative (N) bit is set:

```
MOV r0, #40
MOV r1, #42
SUBS r2, r0, r1 ; 40 - 42, the result is negative
BMI destination
; this portion of code is never executed
```

PL — Plus

This condition is true if the Negative (N) bit is cleared. This happens when a mathematical operation results in a positive number, but also when the result is zero. (Zero is considered positive.)

CS — Carry Set

The Carry Set flag is set when an operation on an unsigned 32-bit overflows the 32-bit boundary.

CC — Carry Clear

The instruction is executed if the Carry Flag (C) is cleared.

HI — Higher

The instruction is executed if the Carry Flag (C) bit is set and if the result is not Zero (Z).

LS — Lower or Same

The instruction is executed if the Carry Flag (C) bit is cleared or if the result is Zero (Z).

GE — Greater Than or Equal

Greater than or equal works on signed numbers and is executed if the Negative (N) bit is the same as the Overflow (V) bit.

LT — Less Than

Less than works on signed numbers and is executed if the Negative (N) bit is different from the Overflow (V) bit.

GT — Greater Than

Greater than works on signed numbers and is equivalent to GE (Greater Than or Equal) and is executed if the Negative (N) bit is the same as the Overflow (V) bit, but also only if the Zero (Z) flag is not set.

LE — Less Than or Equal

Like LT (Less Than), this condition is executed if the Negative (N) bit is different from the Overflow (V) bit, or if the Zero (Z) flag is set.

Comparison of the Different Conditions

Table 4.1 lists the different condition codes and shows exactly which condition flags are used.

TABLE 4.1: Condition Codes

CODE	MEANING	FLAGS
EQ	Equal equals zero	Z
NE	Not equal	!Z
VS	Overflow	V
VC	No overflow	!V
MI	Minus/negative	N
PL	Plus/positive or zero	!N
CS	Carry set/unsigned higher or same	C
CC	Carry clear/unsigned lower	!C
HI	Unsigned higher	C and !Z
LS	Unsigned lower or same	!C or Z
GE	Signed greater than or equal	N == V
LT	Signed less than	N != V
GT	Signed greater than	!Z and (N == V)
LE	Signed less than or equal	Z or (N != V)
AL	Always (default)	Any

Updating Condition Flags

By default, data processing instructions do not update the condition flags. Instructions update the condition flag only when the S flag is set (`ADDS`, `SBCS`, and so on). The exception to this rule is comparison operations, which automatically update the condition flags without the need to specify S.

Consider this code:

```
MOV r0, #0x8000000F
MOV r1, r0, LSL #1
```

In the first instruction, you can put the value `0x8000000F` into the register `r0`. In the second instruction, you can move that value to `r1`, after having performed a left shift by 1 bit. This operation is shown in Figure 4-3.

FIGURE 4-3: Result of a barrel shift

By performing a left shift, the value held in `r0` was read in, and then its value was changed by the barrel shifter to `0x1E`. Bit 31 was shifted left, effectively leaving the scope of a 32-bit number and was discarded. Bits 4, 3, 2, and 1 were shifted to bits 5, 4, 3, and 2, and a new bit 1 was inserted, or "padded" as a 0, as specified by the LSL instruction. You didn't ask for a status update, so you didn't get one. The condition flags in the CPSR remain unchanged. Now look at what would have happened if you had specified the S flag:

```
MOV r0, 0x8000000F
MOVS r1, r0, LSL #1
```

Just like before, you insert the value `0x8000000F` into `r0` and then use the barrel shifter. Just like before, bit 31 leaves the 32-bit scope. Because you are currently working in unsigned 32-bit numbers, the result is considered to be a Carry; the C flag of the CPSR is updated.

By performing manual updates to the CPSR condition flags, you can now execute conditional instructions. You can also execute several conditional instructions if you take care not to modify the CPSR again. After this calculation, you could, for example, have branch instructions depending on several factors. Was your value zero? This is equivalent to a `BEQ`. No, your result was not equal to zero, so this would not result in a branch. Maybe afterward you would do some quick calculations, and so long as you don't specify the S flag (or you don't execute a compare operation), the CPSR condition flags remain unchanged. However, on the next line, you could have a Branch if Carry Set, or `BCS`, and this time you would branch. Because the CPSR hasn't been modified since your last `MOVS`, you can still use the results many lines of code later. This is one of the strong points of ARM; a single calculation can be tested several times.

Now look at a more complete example:

```
MVN r0, #0
MOV r1, #1
ADDS r2, r0, r1
```

The first instruction, `MVN`, is a special instruction that moves a negated number to the specified register. By writing the inverse of 0, you are actually writing `0xFFFFFFFF` to the register. The reasons for this will be explained later. For now, don't worry about the instruction; just remember that `r0` contains `0xFFFFFFFF`.

The second instruction moves the value 1 into `r1`.

The final instruction is an `ADD` instruction; it simply ADDs the content of `r0` and `r1`, and puts the result into `r2`. By specifying S, you can specify that you want to update the CPSR condition flags. Now add `0xFFFFFFFF` and `0x1`, resulting in `0x100000000`. A 32-bit register cannot contain this number; you have gone further than is possible. The logical addition result is held in `r2`; the result is `0x0`.

The result is 0, which is considered to be positive, and so the N (negative) bit is set to 0. Because the result is exactly 0, the Z (zero) bit is set. Now you need to set the correct values for the C and V bits, and this is where things get tricky.

If you are talking about unsigned 32-bit numbers, then the result exceeded 32 bits, so you lost some data. Therefore the C (carry) bit is set.

If you are talking about signed 32-bit numbers, then you essentially did a –1 + 1 operation, and the result is zero. So even though the answer exceeded the 32-bit boundary, the answer did not overflow (meaning the answer did not exceed a signed 32-bit value), and therefore the V (oVerflow) flag is not set.

It is essential to know exactly what sort of result you are expecting. Carry and Overflow do not show the same thing, and the condition codes you specify need to be precise.

Addressing Modes

In ARM assembly, you invariably need to fetch data from one place and put data in another. Your system could just take a set of predefined numbers and do some calculation on them, but that would have severely limited use. Instead, a typical system is constantly fetching data from memory, or from external components (a pressure sensor, a keyboard, or a touch screen, for example).

In assembly, you have several ways of specifying where you want your data to come from. Don't worry too much about the actual instructions yet, more detail will be given in Chapter 7, "Assembly Instructions," but for now, concentrate on two instructions. `MOV` moves data from one register to another, and `LDR` loads data from a specific place in memory to a register.

One of the most common things that you can do is to put a value into a register. You can do this with an immediate value. An immediate value is an integer but only one of a certain type. For this example, use a simple value, one that is an immediate value. To specify an immediate value, put a hash sign in front of the number, like this:

```
MOV r1, #42
```

In this case, tell the processor to put the value 42 into `r1`.

In some cases, you want to move data from one register to another. This is a simple case and can be specified like this:

```
MOV r1, r0
```

This command "moves" the contents of `r0` into `r1`. Converted to C, this is the equivalent of `r1 = (int)r0`. Technically it is a copy and not a move because the source is preserved, but you look at that more closely later in the chapter. By specifying two registers, you simply copy the value from one to another. However, in some cases, you want to do something called a shift. Shift operations are done by the barrel shifter; more information is available in Chapter 7, "Assembly Instructions." A shift takes a binary number and "shifts" the bits to the left or to the right.

Shifting is a quick way to multiply or divide by powers of 2 or sometimes to read in only a portion of a number. It takes the binary value and "pushes" the numbers in one direction or another, increasing or decreasing by a power of two. This is illustrated in Figure 4-4, where 0100 in binary (4 in decimal) is shifted left, becoming 1000 in binary, or 8 in decimal.

FIGURE 4-4: Binary shift left

To `MOV` a register after performing a shift, use the `LSL` or `LSR` operand.

```
MOV r1, r0, lsl #2
```

Like the previous instruction, this command takes the value in `r0`, and puts it into `r1`; however before doing that, it performs a left shift of the number by 2 bits. In C, this translate to `r1 = (int)(r1 << 2)`. It is also possible to shift a number to the right:

```
MOV r1, r0, lsr #4
```

This is the power of ARM assembly, and one of the reasons why ARM systems are so powerful. Now have a close look at what you have done. You have read in a value from a register, performed a shift, and then put the result into another register. This was all done in one instruction.

`LSL` and `LSR` are not the only instructions that you can use; for a complete list, please see the "Barrel Shifter" section in Chapter 7, "Assembly Instructions."

What happens if you don't know exactly how much you need to shift? ARM assembly again comes to the rescue; you can specify the contents of a register to perform your shift:

```
MOV r1, r0, lsr r2
```

By specifying a register for your shift, `r0` can be shifted by the value contained in `r2`.

So now you know how to specify registers and how to put arbitrary values into registers, but it doesn't stop there. However, `MOV` can put values only from registers or from immediate values into registers, so for the rest of this section, you will have to use another instruction, `LDR`. `LDR` reads data from the system memory and puts the result into a register:

```
LDR r1, [r0]
```

By putting `r0` in square brackets, you tell the compiler that you want to get the value of the memory address stored in `r0`. For example, `r0` might contain `0x101E4000`, which is the GPIO1 interface on an ARM Versatile board. Executing this instruction can make the processor look at the memory pointed by `r0` and put the result into `r1`. This is illustrated in Figure 4-5.

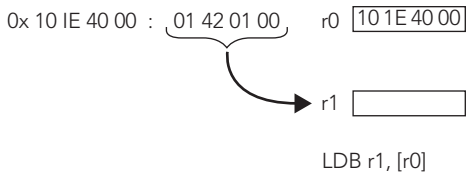

FIGURE 4-5: Loading a register from pointer

To get the memory contents stored at the address in `r0` with an offset, add a number, for example:

```
LDR r1, [r0, #4]
```

This works in the same way as your previous example, except now the processor fetches the memory location `r0` + 4, or in C, `r1 = *(r0 + 1)` because you read in 32 bits. This is illustrated in Figure 4-6.

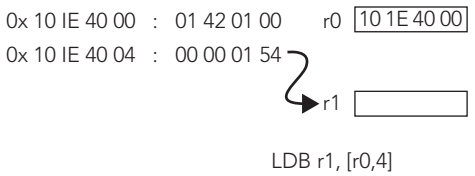

FIGURE 4-6: Loading a register from pointer with offset

Of course, just like before, you can also specify an offset not only as an immediate value, but also as the contents of a register:

```
LDR r1, [r0, r2]
```

This instruction fetches the memory location `r0`, plus the offset `r2`. Shifts can also be used directly in the instruction, like this:

```
LDR r1, [r0, r2, lsl #2]
```

This loads `r1` with the memory located at `r0`, plus the value in `r2`, divided by 2. The equivalent in C is `r1 = *(r0 + ((r2 << 2) /4))`. This gives you the possibility to read, for example, a string of characters from memory, using `r0` as the base address and `r2` as an offset. Of course, after that, you would have to increase the offset register, but there are ways of doing that automatically:

```
LDR r1, [r0], #4
```

Just like in a previous example, this instruction takes the data held in the memory location pointed by `r0` and places it in `r1`. However, the immediate value 4 is then added to `r0`. Consider this example:

```
MOV r0, #200   ; Put 200 into r0
LDR r1, [r0], #4   ; Reads in memory location 200, then r0 = 204
LDR r1, [r0], #4   ; Reads in memory location 204, then r0 = 208
```

This is known as post-index addressing because you have your index, and after having used it, you increment the value. Pre-index addressing works on the same principle and is designated by an exclamation mark:

```
LDR r1, [r0, #4]!
```

This increases `r0` by 4 before fetching the memory. Let's look at another example:

```
MOV r0, #200   ; Put 200 into r0
LDR r1, [r0, #4]!     ; r0 = 204, then reads in memory location
LDR r1, [r0, #4]!   ; r0 = 208, then reads in memory location
```

ARM ASSEMBLY PRIMER

Like any programming language, Assembly can be a little confusing when starting, and like just about any programming language, there are different dialects, or different ways of writing the same thing. The current standard is known as Unified Assembler Language (UAL) and is a common syntax for both ARM and Thumb (which is discussed in Chapter 7, "Assembly Instructions").

Loading and Storing

Essential to any calculation, data must first be loaded into one or several registers before you use it. ARM cores use a load/store architecture, meaning that the processor cannot change data directly in system memory; all values for an operand need to be loaded from memory and be present in registers to be used by instructions.

There are only two basic instructions used to load and store data: `LDR` loads a register, and `STR` saves a register.

Setting Values

Frequently, you need to update a register with a particular value, not something located in memory. This is useful when comparing data. Is the value of the register `r0` equal to 42? You can also use it when writing specific data into a device register; for example, place the data `0x27F39320` into the DDR-II control register to activate system memory.

Branching

Branching is the power of any processor; the capacity of running segments of code depend on a result. It is a break in the sequential flow of instructions that the processor executes.

There are two types of branches possible: relative and absolute. A relative branch calculates the destination based on the value of the PC. Relative branches can be in the range of +/– 32 M (24 bits x 4 bytes) for ARM, and +/– 4 M for Thumb. Because branch instructions are PC-relative, the code generated is known as relocatable; it can be inserted and run at any address.

Absolute branching works differently. Absolute branches always jump to the specified address and are not limited to the +/– 32 M barrier. They use a full 32-bit register, so this value needs to be entered before, costing cycles, but the advantage is that you can access the full 32-bit address range.

Conditional branching is the basis of every system. A computer is not a computer if it cannot be told to do one thing or another, depending on a previous result. Understanding branching is vitally important.

Branching can be done by linking, thereby saving the next instruction address, allowing the program to return to the exact same location after executing a subroutine. Branching can also be done without saving the link register, which is often used during loop instructions.

All processors can branch, but ARM systems take this a step further. ARM cores can execute either ARM assembly instructions, or Thumb instructions, and switching between the two is as easy as issuing a branch and exchange instruction.

All the jump instructions are detailed later in Chapter 7, "Assembly Instructions."

Mathematics

Because every value inside a processor is a number, everything that is done to that number is in some way mathematical. A graphical user interface consists of lines and rectangles, and resizing windows often involves manipulating numbers. Listening to digital music often involves heavy and repetitive mathematics. ARM cores contain a complete instruction set that can handle just about any calculation required for low-end microcontrollers all the way to advanced application processors.

Assembly instructions attempt to be readable; `MUL` is short for multiplication, `SUB` subtracts, and `SBC` subtracts with carry, and in all cases, the variables are in human-readable format.

Understanding an Example Program

Now look at an example program, without having had a look at all the instructions available. This is a mystery routine, and all that is known is that it accepts a single parameter: `r0`.

```
sum
     MOV r1,#0
sum_loop
     ADD r1,r1,r0
     SUBS r0,r0,#1
     BNE sum_loop
sum_rtn
     MOV r0,r1
     MOV pc,lr
```

At a glance, this is an easy routine, but it doesn't make much sense. Now break that down into several sections:

```
sum
     MOV r1,#0
```

This portion of code "moves" the value 0 into `r1`. Presumably, you use `r1` during a calculation, and this is just to set the parameter. In C code, it would be the equivalent of `int x = 0`.

```
sum_loop
     ADD r1,r1,r0 ; set sum = sum+n
     SUBS r0,r0,#1 ; set n = n-1
     BNE sum_loop
```

The first instruction adds together the values held in `r1` and `r0` and puts the result in `r0`. The second line is a subtract instruction, `SUB`, but because the S is present at the end of the instruction, it also updates the condition flags of the CPSR. The instruction takes the value 1 from the value held in `r0` and puts the result back into `r0`. So, `r0 = r0 - 1`, or, in C, `r0--`. The third instruction is a branch operation, making the execution "jump" to a specific location, but only if the NE condition is met. So, this instruction jumps back to the beginning of the code if `r0` is not equal to zero.

So, `r0` holds a value, and `r1` equals `r1` plus the value held in `r0`. Then, subtract 1 from `r0`, and repeat the process while `r0` isn't equal to 1. If you started off with the value 5, the operation would have been 5 + 4 + 3 + 2 + 1, before continuing. In other words, this routine takes a number *n* and returns the result of 1 + 2 + 3 + … + *n*:

```
sum_rtn
     MOV r0,r1
     MOV pc,lr
```

So, what happens here? In the first instruction, the register `r1` is "moved" into `r0`. In the second instruction, the Link Register is "moved" into the Program Counter, but why? ARM functions return their result in `r0` so that is why the temporary register, `r1`, must be first copied into `r0`; otherwise the result would be lost. As for the Link Register, it is a way for returning from a Branch with Link, or a specific way of calling a subroutine. This program was a subroutine, and now it returns back to the main program after completing its task.

Congratulations; you have just survived your first ARM assembly program!

SUMMARY

In this chapter, I have given a brief introduction to ARM Assembly, its uses and applications, and a brief introduction to some of the instructions and options that make ARM assembly unique. You have considered the different condition codes that make most ARM instructions conditional, and I explained what makes this so powerful. I have also shown an example program in assembly, and you've seen that it isn't too difficult to understand assembly.

In the next chapter, I will give a few example applications, from the simplest emulated program to two real-world programs using evaluation boards.