



---

## Chapter 6: Introduction to ARM Memory and Assembly Language

---

In this chapter we jump into the core (pun intended) of MPG and examine the very low level details of the ARM processor. We start by looking at the memory structure in the ARM7 and understand how registers, peripheral special function memory, RAM and ROM work together in the microcontroller. Then we will examine the full instruction set on the ARM7TDMI core and focus in on the instructions that are of greatest interest and most commonly used.

### 6.1 ARM7 Memory Space

Before writing code, you must understand the system that you will be working with. Writing code in high-level languages on PCs often provides the luxury of removing the necessity of knowing how memory in the system is being used not to mention what is actually happening on the bit-level of the microcontroller. The operating system and compiler tends to take care of RAM allocations, you know your program lives somewhere on the hard disk, and any time you need to make use of hardware you can call an API or function in a .dll. You probably do not have a clue about how all the hidden code works and what the operating system is doing in the background – and that is perfectly fine.

In MPG and in small embedded systems in general, you DO need to be aware of all the details like this. It often becomes up to the embedded designer to manage memory and know exactly what resources are available and how they are used in the system. It is not uncommon to build a system from scratch, which means you will have to write all of the drivers that your higher-level functions will call. Though it might sound a bit daunting, microcontrollers are setup and documented to make the process go quite smoothly. If you build your system of drivers intelligently, you can re-use them for different applications running the same processor.

The first step is to know the definitions between the different memory resources that you have available. Registers, RAM, Flash, cache – all these terms refer to different places where data will live and can be accessed in various ways. You also must learn things like how fast accesses to different memory locations are, and whether or not your data will be volatile or non-volatile.

#### 6.1.1 Core Registers

Registers are volatile memory spaces that have specific purposes, though the implementation of a register can vary between processors or even vary within the same microcontroller. Some registers are connected to the processor bus and are addressed with pointers to access the data they hold. In essence, they are RAM locations with special functions. Other registers may live entirely inside the processor core and are accessed in a single clock tick during the execution phase of the instruction cycle.

In the ARM7, there are 17 core registers available in the standard mode of operation, and another 20 registers for the other operation modes that become available at different times. Figure 6.1.1.1 shows the complete list of registers for all the ARM modes.

System and User	FIQ	Supervisor	Abort	IRQ	Undefined
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	r8_fiq	r8	r8	r8	r8
r9	r9_fiq	r9	r9	r9	r9
r10	r10_fiq	r10	r10	r10	r10
r11	r11_fiq	r11	r11	r11	r11
r12	r12_fiq	r12	r12	r12	r12
r13	r13_fiq	r13_svc	r13_abt	r13_irq	r13_und
r14	r14_fiq	r14_svc	r14_abt	r14_irq	r14_und
r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)

ARM state program status registers					
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und


 = banked register

Figure 6.1.1.1: ARM state registers

Source: LPC214x User Guide UM10139 Rev. 3 - 4 October 2010

The system modes are discussed a bit more later on, but for now what you need to understand is that some registers are common to all modes while others are unique. As annotated above, there are some registers that are “banked.” Any banked register is unique to its particular state and would actually be a different physical memory location even though the instruction address to write to it would be the same regardless of mode. For example, if you wanted to write to r13 in whatever mode, you would use r13 = some\_value, and not actually specify the unique name r13, r13\_fiq, r13\_svc, etc. So if you wrote a value into r13 while in User mode then switched to FIQ mode, the value in r13 User mode would not be available. While in FIQ mode, you could write to register r13 again and not impact the value you wrote during user mode.

On the other hand, non-banked registers are shared, so regardless of what mode you are in if you write a value to, say, r0 while in User mode and then switch to FIQ mode, the value you wrote is still available in r0 and would be overwritten if you chose to write to r0 while in FIQ mode. It is as if “mode” controls a bus multiplexer that for some registers will select a different location. Figure 6.1.1.2 might help clear that up a bit.

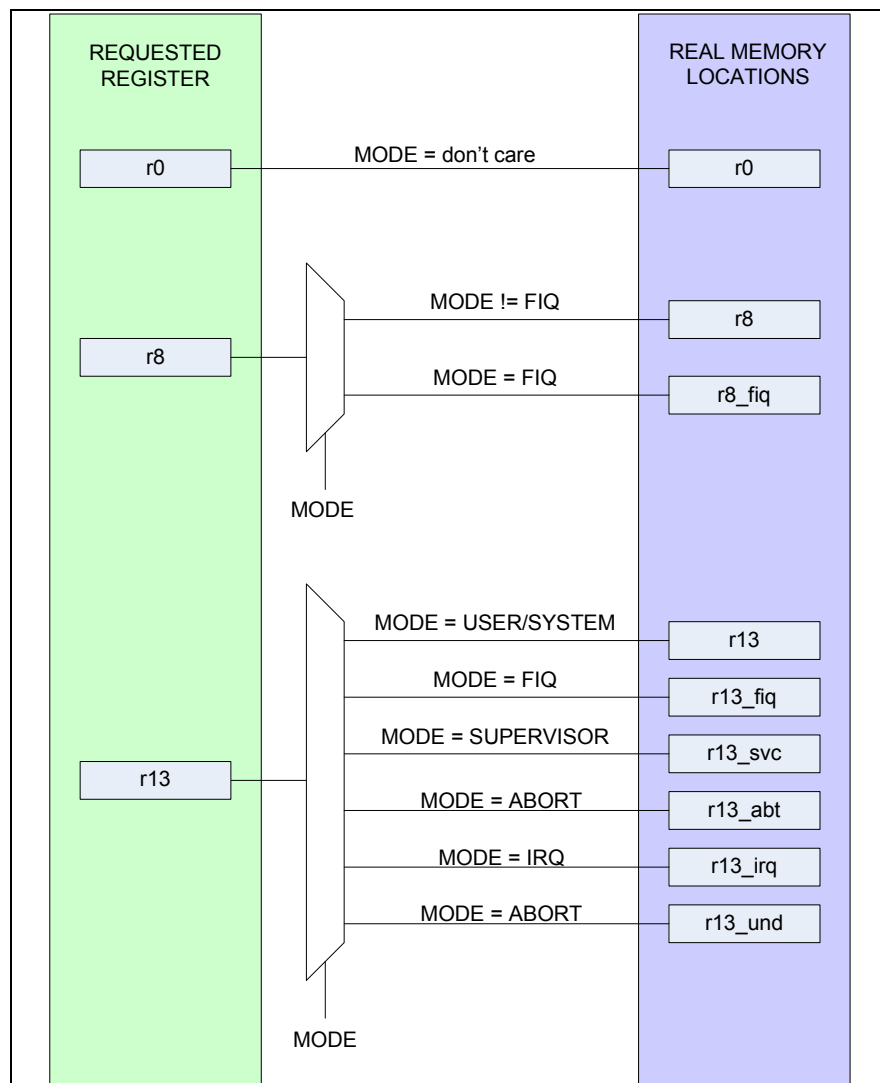


Figure 6.1.1.2: representation of memory accesses to banked registers

For a banked register, your code would always specify the same register to access, but the current mode of the processor would direct that access like a multiplexer to select the value particular to the mode the core is currently in.

You will use the processor in System or User mode most of the time. Here, the first 16 registers are general purpose and used to hold data or addresses that will be used with arithmetic or logical operations. Of these, the first 13 are generic and denoted r0 thru r12. The registers r13

and r14 – though technically available to use as you wish – are understood to have special functions and will be used by the compiler for special functions (so if you start writing to them in an assembler file and link compiler-generated code into you project, you will get some nasty results). Register r13 is reserved for a stack pointer. Register r14 is the “link” register that is used to hold return addresses for function calls. Register r15 is the program counter regardless of whether you are writing in assembler or C. It will continually be updating automatically from hardware and really cannot be used for anything else.

If you find all of that confusing, the good news is that if you are programming in C or C++, you generally do not need to worry about it. In fact, we will never look at this again in this course as soon as the assembly language chapters are done!

The last core register to look at is for processor status and is called the Current Processor Status Register (CPSR). The CPSR is the main status register that holds flags to feedback the status of instruction results, enable/disable interrupts globally, and holds the processor mode. It is probably safe to say that every microprocessor / microcontroller has a status register very similar to this. Unlike the other registers, the CPSR register is updated regularly by the processor as a program runs. In total, 12 bits out of the 32 bits available in the CPSR are used for flags as shown in Figure 6.1.1.3.

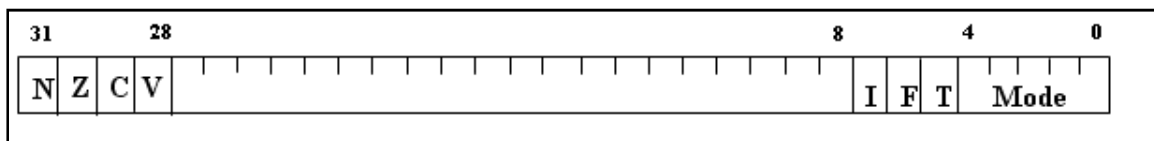


Figure 6.1.1.3: Bit assignments in the CPSR

The bits are described as follows:

*Status bits / condition code flags:*

- **N: Negative flag.** A negative result occurred from the last instruction (e.g. the result has bit 31 set)
- **Z: Zero flag.** The last instruction resulted in 0 (e.g. subtract two registers  $r1 - r2$  where  $r1 = r2$ )
- **C: Carry flag.** A carry, borrow or extend occurred out of the MSB (bit 31) in the last operation (e.g.  $r1$  was shifted left when  $r1$  held 0xf000 0000, or two large numbers were added where the result was larger than 32 bits)
- **Overflow flag.** An overflow into the sign bit occurred (e.g. add two registers  $r1 + r2$ , where the resulting sum is greater than 0xffff ffff)

You will likely find similar status bits on any micro you use, as they are fundamental to how a micro operates and “makes decision” in your program.



*Control bits:*

- I: set to disable interrupts
- F: set to disable fast interrupts
- T: cleared to keep processor in 32-bit ARM state (vs. 16-bit THUMB state)
- Mode: 5 bits to define one of the 7 processor modes of operation (User, Fast Interrupt, Interrupt, Supervisor, Abort, System, Undefined)

The remaining bits are reserved (unused on this core, but potentially used in future designs). Making use of CPSR will be described later in this chapter and other chapters. The SPSR\_x registers in the other modes are “Saved Processor Status Registers” that hold a copy of the CPSR from whatever previous mode the processor was operating in. Part of “mode switching” involves the saving and restoring of the CPSR using SPSR registers. The registers are organized like this because of the way CPSR is implemented (i.e. there are logic paths in and out of the various bits that let them get set or let them control other logic in the processor). It would be too difficult to route these paths and achieve the necessary behavior using banked registers like the other core memory locations use.

### **6.1.2 Peripheral Registers / Special Function Memory**

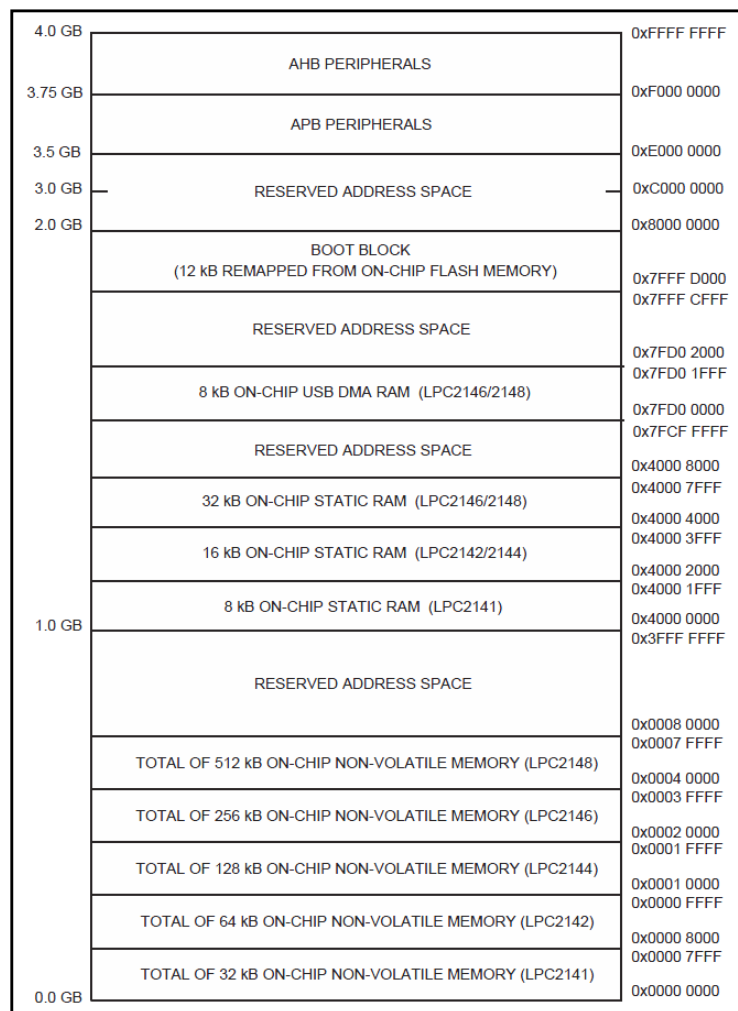
There are a bunch of volatile memory locations that are at specific addresses used by peripherals attached to the core on the address and data bus. When discussing these memory locations, the term “register” is usually used, even though these differ from the core’s registers since they are not directly accessible to the core bus. Any operation involving these memory locations will generally have to load the location data from memory to a processor register, operate on the data, and then store it back.

The reason that these memory locations are considered “special” is that reading and writing values to them will make things happen in the microcontroller. Each peripheral register has some associated logic attached to it, thus the contents of the register will impact what the peripheral does or can be written by the peripheral to tell you what it is doing. If you are addressing memory space directly, you need to be very careful to not play in the peripheral address space accidentally. Unless you are running an operating system that will monitor the addresses you are trying to use, any memory location can be touched by your code which can cause results ranging from practically nothing or mildly weird, all the way up to completely crazy and disastrous!

Manufactures that employ an ARM7 core add peripherals as they see fit to build up their microcontrollers. The core itself does not care about what is attached to its bus, nor does it care if reading or writing to a specific address will do something special. The communication protocols to talk to peripherals are very clearly defined so as long as the protocol is adhered to,

the microcontroller manufacturer can attach anything they want (this is on the device silicon, all internal to the microcontroller).

The manufacturer also has the freedom to specify the addresses of their peripheral special function memory locations, though by definition there is a range of addresses in which those peripheral registers must be located. These locations hold the data that defines how a peripheral runs, and would be described in the datasheet for the particular microcontroller. Figure 6.1.2.1 shows the memory regions defined for the LPC214x family of processors.



*Figure 6.1.2.1: LPC214x family system memory map.*

*Source: LPC214x User Guide UM10139 Rev. 3 - 4 October 2010*

All of the peripherals (except the software interrupt peripheral) on the LPC214x processors exist in the “VPB” (VLSI Peripheral Bus) address range. From Figure 6.1.2.1, you can see that addresses start at 0xE000 0000 and end at 0xEFFF FFFF. Though this is a quarter of a GB worth of addresses, the LPC214x family incorporates space for just 2MB of addresses, which is

allocated into 128 possible peripherals at 16kB of space each. The mapping for the LPC214x peripherals is shown in Figure 6.1.2.2.

APB peripheral	Base address	Peripheral name
0	0xE000 0000	Watchdog timer
1	0xE000 4000	Timer 0
2	0xE000 8000	Timer 1
3	0xE000 C000	UART0
4	0xE001 0000	UART1
5	0xE001 4000	PWM
6	0xE001 8000	Not used
7	0xE001 C000	I <sup>2</sup> C0
8	0xE002 0000	SPI0
9	0xE002 4000	RTC
10	0xE002 8000	GPIO
11	0xE002 C000	Pin connect block
12	0xE003 0000	Not used
13	0xE003 4000	ADC0
14 - 22	0xE003 8000 0xE005 8000	Not used
23	0xE005 C000	I <sup>2</sup> C1
24	0xE006 0000	ADC1
25	0xE006 4000	Not used
26	0xE006 8000	SSP
27	0xE006 C000	DAC
28 - 35	0xE007 0000 0xE008 C000	Not used
36	0xE009 0000	USB
37 - 126	0xE009 4000 0xE01F 8000	Not used
127	0xE01F C000	System Control Block

*Figure 6.1.2.2: LPC214x family peripheral mapping.*

*Source: LPC214x User Guide UM10139 Rev. 3 - 4 October 2010*

Remember that these addresses are microcontroller-specific. For an example from the LPC214x family, the base address for the Timer 0 peripheral is 0xE000 4000 as you can see from Figure 6.1.2.2 (you do not need to know what a timer peripheral does yet!). Another manufacturer who wants to include a Timer 0 peripheral and is also using the ARM7 core for its



microcontroller may choose to declare Timer 0 at address 0xE0008000. It does not matter beyond the required awareness of where to set the timer0 symbol mnemonic (and this is probably done for you in a header file).

So what does all of this address talk actually mean? As was eluded to, peripherals require configuration data to know how to run and to report back status and other data. Looking back at the Timer 0 example, it probably needs some setup registers to tell it where to start and stop, how fast to run, whether or not it should interrupt the processor, etc. You would also want to have access to the current count that the timer is keeping among other things. The special function memory locations give you these specifications. From the LPC214x User manual (UM10139 Rev 3 – 4 October 2010, Table 238, page 247), we can see that:

- 0xE000 4000 is the Timer 0 interrupt register, T0IR
- 0xE000 4004 is the Timer 0 control register, T0TCR
- 0xE000 4008 is the Timer 0 timer counter register, T0TC

There are 14 other registers related to Timer 0 as well, and depending on how you are using the peripheral, you may need some of these though often you can ignore many of them depending on how you are using the peripheral. Configuring and using peripherals will be covered in some of the next chapters, but for now just understand that certain memory locations that you write to will result in different things happening within your microcontroller, so always be aware of the addresses you specify in your instructions.

### 6.1.3 General Purpose RAM

The next group of memory to discuss is general purpose RAM. These locations are not linked to any special operation of the device or its peripherals. They are entirely available for the programmer to use in a program as required. Depending on the family of processors and specific device within that family, the amount of available RAM will vary. In many cases, ARM microcontrollers can address external RAM so you can drastically increase the amount of memory available to your system. That being said, the RAM available on the LPC214x used in MPG will be more than enough. RAM starts to become an issue when you are dealing with a lot of external communication like over a USB bus or transferring data to an SD card. Even then, a system with 16kB of RAM is still very capable of creating complex devices, so no worries. If you get into sampling audio or working with graphical LCDs like on a cell phone, then RAM becomes much more of a concern with your system. No worries in this course, though!

In the context of RAM, think of each location as being a 32-bit chunk of memory available for you work with. There are times when you can focus on specific sections of a 32 bit number (8-bits, 16-bits), but you have to be careful that you do not misalign yourself. Addresses always refer to the start of a byte in memory, so 0x100 and 0x101 are 8 bits apart in memory. By extension, 32-bit address are always 4 addresses apart, so when working with two consecutive 32 bit numbers, the first would live at address 0x100 and the second would live at 0x104. This can burn an embedded programmer working in assembler where pointers are used and not





correctly incremented. You probably did a lab in school where you examined the difference between incrementing a pointer-to-char and a pointer-to-long. The compiler will take care of the simple operation `pointer++` and increment the char pointer by one byte and the long pointer by 4 bytes. If you are managing that pointer on your own in assembler, you have to make sure you increment it correctly.

RAM addresses start at 0x40000000 on the LPC214x. Somewhere in all that space will be your function stack or stacks (depending on how complicated you are getting), heap, and global variables. If you write code in C, then most of the memory management is taken care of by the compiler, though there may be situations where you will want to specify groups of memory or data structures that occupy specific locations. For example, you would want a transmit data buffer to occupy a sequential block of memory so you would be able to loop a pointer through to read out the data quickly.

In general, it is a good idea to know how your memory is managed in an embedded system. The most important thing to consider is running out of memory as that is a very real possibility on an embedded system with limited resources. A compiler would catch an error where too much space was allocated for a variable (for example, if you tried to allocate a 1MB data buffer). Probably the most dangerous memory problem is with stack overflows, where you get a few too many function calls in the pipeline and end up taking more stack space than what you have available. In this scenario, your system would be toast, though the processor can be set up to detect such an event and make an attempt to recover or reboot, but it is definitely not an error handler you ever really want to hit!

### 6.1.4 Flash memory

The last type of memory on board the processor is non-volatile flash memory. This is where your program is stored and will always be retained even when power is taken away from the chip. It is still electrically written and erased, but is done so in a special way that essentially forces charge into ultra-high impedance memory cells and locks it away there forever. Well, forever is overstating it –programmed flash memory is usually rated for at least 40 years of data retention.

Flash memory can be read quickly so a program can run quickly, but it cannot be written very fast. Writing flash memory also requires a high programming voltage like 12V or higher in order to cram charge in or force it out. Most processors these days have built-in charge pumps to generate that programming voltage when told to do so by whatever programming system is writing to them. Other processors require the programmer itself to generate the high voltage. If the processor does have the ability to achieve a voltage high enough for programming, then a program you write can be capable of programming itself as long as you give it the information with which to do that. The type of program is often called a “boot loader,” which is a small program designed to reprogram a large chunk of memory. The boot loader never gets erased or overwritten, but the memory locations for main program can be erased and loaded as required.



The LPC2142 has 64kB of flash memory space and that can go as high as 512kB on the highest end member of the family, LPC2148. The addresses begin at 0x00000000 and extend to 0x0007FFFF (512,000 bytes). Like in our discussion of RAM size, any memory less than 1 MB seems very small when compared to desktop systems that have terabytes of disk storage and gigabytes of RAM. But also like the case of RAM, huge volumes of flash storage are not needed as the embedded system tends to be far less complex, far more targeted, and does not have to include memory intensive graphics, operating systems and user interfaces. You will be amazed at how much you can do with these “limited” resources.

## 6.2 The ARM7TDMI Instruction set

Now that you know a bit about the processor’s memory structure, you can take a look at the instruction set that will be used to make up your program in flash, and execute to allow access to those volatile memory locations. ARM architecture is essentially a customized RISC (reduced instruction set computer) device. Also to note for the sake of being complete, is that the ARM7 core has a Von Neumann-style architecture (versus a Harvard-style architecture). Von Neumann architecture means that both data and instructions use the same bus, vs. Harvard architecture where data would flow across its own bus independent of the instructions. Harvard architecture is actually a bit better (and low and behold, the next version of the ARM core chose it!), but the ARM7 is still a great core that operates very efficiently.

Without going into the comparison details between a RISC and CISC (complex instruction set computer), it can be stated that the ARM architecture relies on a relatively small set of instructions, each of which are very specific and simple in nature. Many of the instructions take only a single-instruction cycle or can be executed in just a few ticks of the clock. For an ARM7, the average instruction length is 1.9 cycles (clock ticks). A RISC processor must combine a series of simple instructions to enable a net result of a more complex operation.

Processor instructions are what your code ultimately becomes once you hit “compile” or “assemble” in the development environment. Even the most dazzling, complex games break down into individual instructions that are stored in memory and executed one at a time (unless you have a multi-core processor, of course). Generally speaking, instructions on a 32 bit core are 32-bits long, and your program can only have as many instructions as the non-volatile memory space allows. Since we know the LPC2148 that has 512kB of Flash space, your program can have a maximum of 128,000 32-bit instructions.

So what makes up the instruction itself? Depending on the instruction type, the 32 bits will be allocated for slightly different things. All instructions include an “op code” (i.e. a few bits that distinguish between the different instruction types). The other bits can be condition codes, register arguments, certain flags, literals and other information. Instructions are unique to the processor core, so other than the fact that instructions for two 32 bits cores have a common length, the bits that make them up will likely be entirely different.



For the ARM7TDMI-S, the core version is ARMv4T – so essentially it is a 4<sup>th</sup>-generation core. The core's name (i.e. ARM7TDMI-S) actually indicates what family and features define it as follows:

- ARM7: The core family
- T: Indicates it supports the 16-bit Thumb instruction set (we will not cover this in MPG)
- D: Indicates the core can use JTAG debugging (this we will use in MPG!)
- M: The core supports fast multiplication
- I: The core has an embedded in-circuit emulator (ICE) macrocell – this is a massively important feature of the ARM7!
- -S: The processor is synthesizable (i.e. you can duplicate it in an FPGA)

As the ARM core matured, more instructions were added and new versions were released. The ARMv4T instruction set has a total of 41 instructions which can be broken down into 5 groups:

1. Data processing instructions
2. Branch instructions
3. Load-Store instructions
4. Software Interrupt instructions
5. Program Status Register instructions

This does not include instructions that can work with a coprocessor, which are excluded here as the LPC214x family does not incorporate a coprocessor. It does include some instructions that are provided for convenience but really are just other instructions with specific operands (i.e. the shift instructions are actually move instructions). If you counted the number of ways an instruction could be modified with various operands, flags, and conditions, there would be well over 100.

The complete list of these instructions along with an important list of notes and definitions is at the end of this chapter. Print these pages and keep them with you at all times when you are writing or debugging assembly code. It will be part of your bible of information and is extremely important! All of the instructions in this list are available on the LPC214x family used in MPG because it uses the ARM7TDMI-S core. Older processors with older cores may not have all the same instructions, and newer processors or more advanced ARM cores may have more. ARM does guarantee backwards compatibility, so assembly code that you write for a version 4 core (ARM7) will run on a version 5 core (like an ARM9), though you would still need to remake the object file for the other processor and link it in properly.



## 6.3 Writing Assembly Code

In the assembly world, programming is quite straight forward. Each instruction type has specific rules that define how it is to be used, because each line of code always gets translated by the assembler into exactly one instruction. With assembly programming, you must be thinking at the bit-level because all you really do with an instruction is move bits around in memory – it is as simple as that. Considering what you can accomplish by moving bits around, it is remarkable that a computing system works. All those incredibly complex programs and games out there all happen because of 32-bits of memory moving from one location to another (mind you, there are an awful lot of moves happening in a short period of time!).

Writing code in assembly is a stark contrast to writing code in a high-level language where it is up to the compiler to understand what the code is trying to do and then translate it into machine language. A compiler actually writes assembly language for the target processor that the code is built for. A single line of C code that is compiled for an ARM core might translate into a single line of assembly, or may end up requiring hundreds of instructions and maybe even some base function calls. It is all based on how the compiler decides to use the instructions available. For example, setting a bit in a register with an OR operation will use an OR instruction, 1-to-1. Trying to do a divide operation on an ARM core (which does NOT have a hardware floating point math processor) would require a LOT of code and memory.

No two compilers are the same, so the same line of C code processed by one compiler may not write exactly the same assembly code as another compiler, though the resulting functionality should match. A simple example would be a bit-clear operation. A smart compiler would make use of an instruction that could clear a bit (one instruction to accomplish the task), whereas a different compiler might go through a processes of loading a bit mask and then logically ANDing the mask and the target register (which would take at least a MOV instruction followed by an AND instruction). To abstract this further, still the same line of C code compiled for an ARM7 core would end up being totally different if it were compiled for a PIC microcontroller since the instruction set is completely different (not to mention that the PIC is an 8-bit micro and the ARM is 32-bit).

Since there is no room for interpretation in writing assembly code, the programmer must ensure that the exact syntax is used for each line of code. Errors will result in one of two scenarios:

1. You write an invalid instruction or use an invalid number of operands – in general, you do something that the assembler does not recognize and therefore cannot translate into an instruction. In this case, you end up with an assembler error that alerts you to the problem so you can fix it.
2. You incorrectly add a flag, prefix, postfix, mix up operands, or specify a literal when you meant to specify an address – anything that is syntactically correct. In this case, the assembler will happily build the 32-bit instruction and make it part of your program,



with results apparent only when your code is running on the target (and hopefully you can catch it!).

The gist is that you must be very careful when putting together your instructions to ensure you are writing what you really want. Commenting your code will be a huge advantage as it will help you and whomever is reading your source code to know what you think you are doing, so what you are actually doing in the line of code can be verified. In fact, when you are learning to write assembly language for any processor, it is a good idea to comment EVERY line of code you write with the C-equivalent, pseudo code, or plain text explanation of what you think you are doing with the assembler.

### 6.3.1 Assembly Language Syntax

Just like in C, there are rules for syntax in assembly and standard ways of formatting lines of code. In assembly language, there are a few less rules because you have a limited number of expressions you can write. Each line of assembly code always has four fields, each of which are tab-delimited from one another: Label, Instruction Mnemonic, Operands and Comments. The format looks like this:

<b>Label</b>	<b>Instruction Mnemonic</b>	<b>Operands</b>	<b>; Comment</b>
--------------	-----------------------------	-----------------	------------------

The Label field is optional, as is the comment field other than the semi-colon. Labels will help organize your code and allow you to branch / jump around in the program because every label actually gets assigned to a corresponding program memory address for the line of code it tags. Comments are just text and are entirely ignored by the assembler, but as already eluded to, their value is extremely high for writing good code.

The Instruction Mnemonic is any one of the 41 reserved words listed in the Instruction set (e.g. ADC, ADD, MOV). Most mnemonics are self-explanatory for what they do, but in case you cannot figure it out, they are described in text in the instruction listing. What is not self-explanatory are the optional flags or conditions that you can set to each instruction. To start an example, look at a simple move instruction that has the mnemonic MOV. Reading the instruction set for MOV, you see the following detail:

<b>Label</b>	<b>Mnemonic</b>	<b>Operands</b>	<b>Comment</b>
move_eg	MOV[cond][s]	Rd, <Op2>	; Move syntax

Any text written in square brackets [ ] is optional, and any text written in angle braces <> is required but has several options. Almost every instruction has a condition field [cond] that makes that line conditionally executable based on the CPSR flags. If you specify a condition, the processor will determine if the instruction should be carried out or skipped. This happens before the instruction is fetched, and does not cost any cycles to evaluate which is actually quite incredible, especially if you have worked with other processors that have to use separate instructions for condition testing. Many instructions also have a flag that tells the processor



whether or not to update the CPSR flags after the instructions has been executed. This is another very powerful feature of the ARM architecture.

From what we have just learned, there are three ways that the MOV instruction can be written:

Label	Mnemonic	Operands	Comment
simple_move	MOV	<operands>	; Move version 1
cond_move	MOV <del>PL</del>	<operands>	; Move version 2
cond_move_s	MOV <del>PL</del> <del>S</del>	<operands>	; Move version 3

The first MOV instruction will move the values specified in the operands and regardless of the result in the destination register, the CPSR flags will not be touched. The second MOV instruction is suffixed with “~~PL~~” which tells the processor to only execute the move if the last instruction that updated the CPSR flags resulted in a non-negative (positive or zero) value. The third MOV instruction is suffixed with both “~~PL~~” and “~~S~~” so it too will be conditionally executed and depending on the result stored in the destination register, will update the N, Z or C flags in the CPSR. This shows that one instruction can have at least 3 variations even before considering the operands and it is very important to correctly specify the mnemonics. Note that order of the conditions and flags is indeed important – MOV~~PL~~~~S~~ is valid whereas MOV~~S~~~~PL~~ is not.

The operands that must be used with each instruction type can be fairly complicated – this is why you must have your instruction set readily available when writing code. In general, you specify a destination register and one or two source registers (or more, depending on the instruction). Building on the example move instruction, look at the operands that are specified for MOV:

Rd, <Op2>

This states that a move instruction requires a specified destination register, Rd, and some sort of second operand, Op2, which must be the source of the data to move. You will see in other instructions registers denoted Rs (source register) and Rm and Rn (other register arguments).

Recalling the registers available, Rd can be anything from r0 to r15, though remember that r13, r14 and r15 are reserved and should not be used in most cases. That leaves r0 through r12 as options. Depending on what you intend to do with the data, you might strategically choose Rd to support the next instruction. For this standalone example, r0 will be used.

The second operand has several optional forms and thus takes a bit longer to explain. Notice that nearly all the Data Processing instructions involve <Op2> as an operand. Fortunately, once you learn all of the variations of Op2 for the MOV instruction, you can apply it verbatim to the others.

Start by looking on the “Notes” page of the instruction set summary in this chapter. There is a block of text under the heading “Operand 2” that describes the 6 options available for Op2 (see



Figure 6.3.1.1). It also describes the conditions that have to be imposed on some of the options. Notice that other than the first case where Op2 is an immediate value, Op2 is actually two arguments. Furthermore, the second option for Op2 (i.e. Rm {, <opsh>}) contains even more options as described by the second half of the table!

<b>Operand 2</b>	
Immediate value	#<imm8m>
Register, optionally shifted by constant	Rm {, <opsh>}
Register, logical shift left by register	Rm, LSL Rs
Register, logical shift right by register	Rm, LSR Rs
Register, arithmetic shift right by register	Rm, ASR Rs
Register, rotate right by register	Rm, ROR Rs
Optional shifts by constant (opsh):	
No shift: Rm (same as Rm, LSL #0)	
Logical shift left: Rm, LSL #<shift> where allowed shifts are 0-31	
Logical shift right: Rm, LSR #<shift> where allowed shifts are 1-32	
Arithmetic shift right: Rm, ASR#<shift> where allowed shifts are 1-32	
Rotate right: Rm, ROR #<shift> where allowed shifts are 1-31	
Rotate right with extend: Rm, RRX	

*Figure 6.3.1.1: Table of Operand 2 options*

If you simply wanted to move a number into a register, the operand is specified as #<imm8m> in the table. The looks complicated, but what it means is that if you wanted to load the value 8 into r0, you would write the instruction like this:

<b>Label</b>	<b>Mnemonic</b>	<b>Operands</b>	<b>Comment</b>
move_eg	MOV	r0, #8	; r0 = d'8'

So why does the table show #<imm8m> instead of just saying “any number?” Well, to make this even more exciting there are conditions on the value imm8m: it must be a 32-bit constant that can be formed by taking an 8-bit number times  $2^{2n}$ , where n is 0 to 12. What? This means that only certain numbers can be loaded directly into a register! It turns out that the numbers available are quite useful and there are not too many times during a regular program that the number you want is not available directly. The full table of numbers that can be generated are shown in the Immediate Table on the notes section of the website.

This might sound crazy, but there are good reasons behind it that we will not be worrying about just yet. The good news is that the assembler will tell you if you have requested a number that is not valid. The bad news is that if you want a number that is unavailable, you need to find another way to do what you want. See section 6.3.2 for more information.



The next choice for Operand 2 is slightly more flexible ( $Rm \{, <opsh>\}$ ) even though it might look just as strange. First, you must specify a register that already has a value in it (you would have loaded this value in a previous instruction). If you want the value as-is, just specify the register name. If you want to shift it by a certain amount that you provide in the instruction, indicate the shift type and amount per the guidelines at the bottom of the table for “opsh.” It is probably a good idea to peek back up at Figure 6.3.1.1 right now to see all these.

The number of shifts is restricted to a maximum of either 31 or 32, as anything more would become redundant on a 32-bit signed or unsigned number. Here are some examples assuming that r1 will be the Rm register:

Label	Mnemonic	Operands	Comment
	MOV	r0, r1	; r0 = r1
	MOV	r0, r1, LSL #4	; r0 = r1 * 16
	MOV	r0, r1, ASR #1	; r0 = signed(r1 / 2)

The last four choices in the Op2 table are probably self-explanatory now. You must have a value in Rm that you want to shift. Then choose the shift type and reference another register, Rs, that holds the number of shifts to make (assumes Rm and Rs have both been loaded previously, where Rm is r1 and Rs is r2; the value in Rs should not exceed the maximum allowed shifts specified). Examples are as follows:

Label	Mnemonic	Operands	Comment
	MOV	r0, r1, LSL r2	; r0 = r1 x 2 <sup>r2</sup>
	MOV	r0, r1, ASR r2	; r0 = signed(r1 / 2 <sup>r2</sup> )

Right now it will not be clear as to why you would need so many similar options (and some that may seem pointless) for a simple move instruction. However, if you start thinking about some more complex operations or scenarios where you are moving numbers around, you might start to see how this can come in handy. Before looking at some other instructions, we will explore the shifts and numbers a bit further.

### 6.3.2 The Barrel Shifter

The ability to shift-on-the-fly by a range of numbers (or references to registers) is done by something called a barrel shifter built on one (and only one) of the data paths into the processor’s arithmetic logic unit (ALU). The barrel shifter can shift numbers in the incoming data word (32 bits) by any amount of shifts in the range 0 – 31 and can do so left or right. Depending on the type of shift specified, it can preserve sign bits and use the carry flag as part of the shifting operation. It is called a barrel shifter due to the way it is implemented – it is NOT a simple digital shift register that requires a clock tick for every shift. Instead, it is more of a two-dimensional matrix that allows the required shift to occur immediately as the data passes along. Perhaps it is more accurate to say the data is selected at a particular position. Yes, there is a delay associated with whatever shift takes place, but this delay is shorter than the overall period of a clock tick and the worst case is accounted for in the 60MHz device maximum.





From Figure 6.3.1.1, there are 5 types of shifts that the barrel shifter can perform: LSL, LSR, ASR, ROR and RRX.

**LSL:** Logical shift left – shifts bits out the MSB and pads 0s to the LSBs. All shifted bits are lost. This also functions as an arithmetic shift left.

**LSR:** Logical shift right – shifts bits out the LSB and pads 0s to the MSBs.

**ASR:** Arithmetic shift right – shift bits right and fill the vacated bits with 0s if the number was positive and ones if the number was negative.

**ROR:** Rotate right – shifts bits right. The LSB rotates to the MSB. Note that there is no **ROL** because you can accomplish any left rotate by x bits by using a right rotate by 32-x bits.

**RRX:** Rotate right extended (through carry) – shifts bits right, where the LSB goes into the Carry flag bit, and the Carry flag bit shifts to the MSB. This is a great way to access each bit in a register if you were, say, sending it out a serial port.

The barrel shifter knows what to do because certain flags are set in the instruction word along with the number of shifts to do. This information is stripped from the instruction word during the decode phase and provided to the barrel shifter so it can setup and do its thing as the data is passed to the ALU for execution.

### 6.3.3 Constant Values

Perhaps one of the strangest concepts of a microcontroller to digest is that of the immediate value. “Immediate” is a fancy term for “number” and is probably named this way because the number is “immediately present” – it is not retrieved from some other memory location. The term “literal” is used as well. In some processors like the 8-bit PIC, immediates are easy because they are at most an 8-bit number and are encoded in the instruction word (which is 14-bits long on a 16-series PIC: 6-bits for the op code and stuff and 8 bits for the literal). So on a PIC, if you wanted the number 248 loaded to a memory location, say 0x80, you would use these two instructions from the PIC instruction set:

```
MOVLW    d'248'    ; Load the working register with 248
MOVWF    0x80       ; Move the value to address 0x80
```

The “MOVLW” instruction on the PIC is “move literal to working register”. The number 248 is part of the 14-bits of the instruction word, so the bit pattern to make 248 (b’11111000’) is actually encoded as part of the instruction in program flash. No register / memory location is larger than 8 bits in the PIC, so any possible number that you can store (0 – 255) can fit in the instruction.

The ARM is similar, but different. Instructions which have the ability to load a number have some space within the 32-bit instruction word to encode a literal, but not enough to encode a full 32 bit number since you need an op code and other bits. So think about it: how do you



“create” a number inside the processor, without providing the entire bit pattern? Some processors have a specific “literal generator” that can be used to “create” a number based on information that can be stored in the instruction word. The ARM makes use of its barrel shifter to do this. We have already talked about what it boils down to: literals that can be loaded into a register can only have certain values, where the allowed values are given by the formula:

$$\text{Immediate} = (0 \text{ to } 255) \times 2^{2^n} \quad \text{where } 0 \leq n \leq 12.$$

When entering these immediates as instruction arguments, you can explicitly state the number you want and the assembler will tell you if what you have chosen is not attainable. You can also specify the literal using the formula with an 8-bit value and “n”, but that is much less intuitive.

What happens if you absolutely need a number that cannot be generated by the formula? For example, what if you needed the speed of light in a calculation (the speed of light is 299,792,458 m/s). This number is NOT available as an immediate generated by the formula, so you have two choices: either create it as an arithmetic combination of numbers that can be generated (at the cost of a couple of moves and an arithmetic instruction), or simply hard-code it in memory in your code (i.e. create a table of values that can be referenced to retrieve the immediate). The latter option is likely the best, as it takes only a single 32-bit FLASH location, an instruction to load its address, and a move instruction to retrieve it. In this way, the number you need is not “generated”, but rather part of the hex file output by the assembler and programmed onto the device, so it is always there. As long as you know where it is, you can go get it and use it – the assembler and linker take care of that for you.

No matter what you do, there is no other way to make electrons inside the processor turn into the literal value you need. The processor is dumb, it does not actually know what a number is. You have to tell it every number you want or make the processor create the number with defined logic like that which generates literals in the ARM. It is such a simple concept that the simplicity makes it difficult and your brain might have trouble accepting it for a while.

## 6.4 Simple Data Processing Instructions

Now that you have the background of the instruction variations, operand types, the barrel shifter, and understand how immediates work on the ARM7, we can look at more instructions and examples of using them. Some of the easiest instructions to work with are the Data Processing instructions, though with so many variations of the instruction and operands, it might be a bit more difficult than first expected. Remember that the ARM always works with numbers in registers, never directly from memory.

We have already looked at some of the examples of a MOV instruction, so we will continue from there. Of the 17 registers that we have at our disposal, we will keep an eye on five of them: r0, r1, r2, and r3 for general purpose registers, and the top nibble (4 MSBs) of CPSR since

that is where the instruction result flags live, denoted NZCV. We will also declare that all of the values we are observing are unsigned (so bit 31 is the MSb and not a sign bit).

As each step is taken, the register contents will be shown. The best way to do this exercise is to download and open up the Chapter 6 firmware in IAR and step through the code as it executes. The code is based on the same framework you looked at in Chapter 5, though you might notice that there are three projects in the Workspace. Close any open files and bring up the first project with chapter6.s as the source code file. Set up a Register and two Watch windows as shown in Figure 6.4.1. Set the four register values in the first Watch window to show in decimal and those in the second Watch window to show in binary (right click Value to choose). You cannot change the hex format in the Register window.

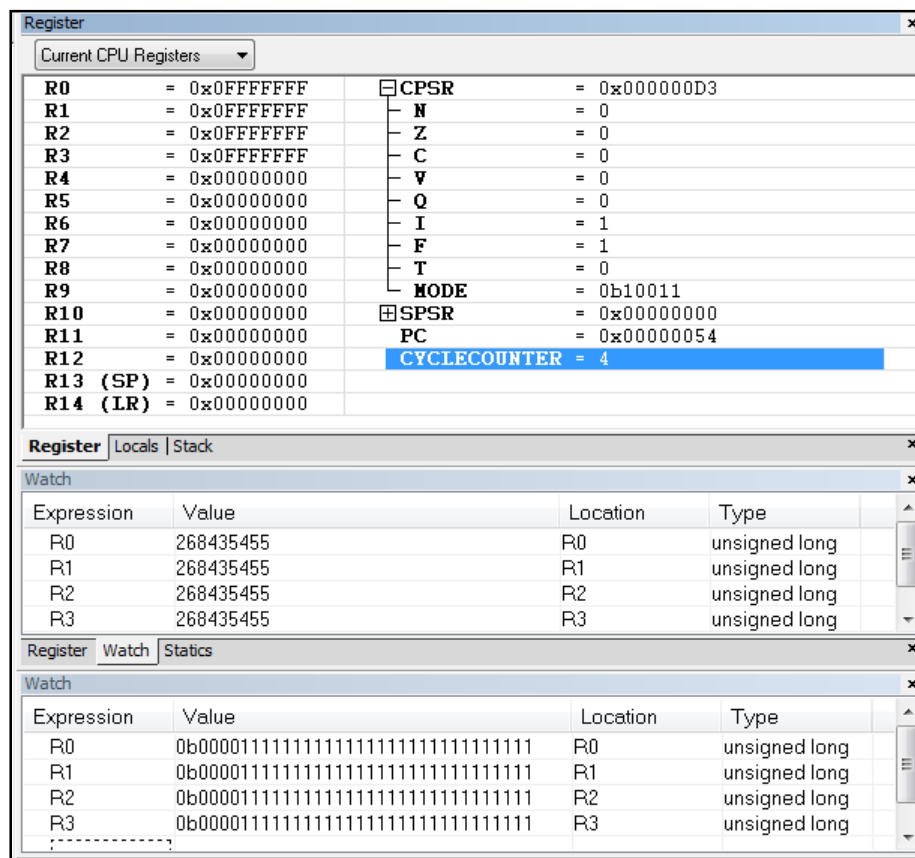


Figure 6.4.1: Debug environment setup Register and Watch windows

As with any uninitialized volatile memory, the start state is garbage, so before we write any code, our memory looks like this:

r0	r1	r2	r3	NZCV
?	?	?	?	?



First, initialize all four registers to 0. Do r0, r1 and r2 like this:

```
MOV      r0, #0          ; r0 = 0
MOV      r1, #0          ; r1 = 0
MOV      r2, r1          ; r2 = r1
```

So the memory looks like this:

r0	r1	r2	r3	NZCV
0	0	0	?	?

Now zero r3 and use the “S” flag to force an update to the CPSR flags:

```
MOVS     r3, #0          ; r3 = 0, update CPSR
```

r0	r1	r2	r3	NZCV
0	0	0	0	0100

Now we have the registers of interest initialized to known values. This of course is exactly what you do in a high level language – get your system in a known state. Now we can safely proceed with the example to use different instructions to change the memory locations and demonstrate the instruction set and processor operation.

Data processing instructions include the following:

- ADD, ADC, SUB, SBC, RSB, RSC for binary arithmetic
- AND, ORR, EOR and BIC for bit-wise logic operations
- MOV, MVN for register moves
- CMP, CMN, TST, TEQ for comparisons
- MLA, MUL, SMLAL, SMULL, UMLAL, UMULL for multiplication
- ASR, LSL, LSR, ROR, RRX for shifts (these are “pseudo” instructions)
- ADR as a pseudo instruction for address loading

It would be a tad tedious to show examples of every one of these, so we will follow through the exercise with an example or two from each of the above groups. There is no point to this code other than to show examples of instruction execution, so do not expect a game of Pong to magically appear if you run this on your ARM – that comes in MPG Level 2!

Right now, all we have are 0s in memory, so we will start by making that a little more interesting. Note in general, the examples will use r0 and r1 as source registers, r2 as a special source, and r3 as a destination – just by our own arbitrary definition. Also to note is that most ARM instructions specify the destination argument first, then the source operands. You should keep your instruction set listing open and be sure you see how the arguments work.



First, fill up r0, r1 and r2 with more interesting numbers. MOV instructions could be used, but we will make life more interesting with some ADDs. It is acceptable to use one of the source registers as the destination register as shown:

```
ADD      r0, r0, #256           ; r0 = r0 + 256
ADD      r1, r1, r0, LSL #2     ; r1 = r1 + (4 x r0)
MOV      r2, r0, LSR #6        ; r2 = r0 / 64
```

r0	r1	r2	r3	NZCV
256	1024	4	0	0100

Now try a subtract with the “S” flag:

```
SUBS     r3, r1, r0             ; r3 = r1 - r0, update CPSR
```

r0	r1	r2	r3	NZCV
256	1024	4	768	x0xx

The other flags will update, but focus on the zero flag for now: the “x” represents don’t care. So the following instruction uses a condition code and will only be executed if the result of the previous instruction (that is, the previous instruction that update CPSR) was zero. Since it was not, this line of code will not actually be executed and r3 will remain 768:

```
MOVEQ    r3, #1024             ; if {Z}, r3 = 1024
```

r0	r1	r2	r3	NZCV
256	1024	4	768	x0xx

Now try a reverse subtraction with a shift by the number in r2. “Reverse” just means that instead of doing operand1 – operand2, the processor will do operand2 – operand1. This is important because operand2 is the only input to the ALU that goes through the barrel shifter, and thus is the only eligible operand for shifts.

```
RSBS     r3, r1, r0, LSL r2     ; r3 = (16 x r0) - r1
```

r0	r1	r2	r3	NZCV
256	1024	4	3072	x0xx



Here is a multiplication instruction conditionally executed as long as the RSB did not result in 0.

```
MULNE      r3, r2, r0          ; r3 = r2 x r0
```

r0	r1	r2	r3	NZCV
256	1024	4	1024	x0xx

Now try a comparison. Comparisons simply “look” at the values in the register to update flags, so no registers will change their value as a result of this instruction other than the CPSR (even though the processor will actually subtract the two numbers). You do not have to specify the “S” flag as it is implied by the instruction that you want to update CPSR.

```
CMP        r3, r1              ; Set flags if r1 == r3
```

r0	r1	r2	r3	NZCV
256	1024	4	1024	0100

For the last example case, use a logical bit-wise OR conditionally executed on the CMP being equal, and update the flags on the result.

```
ORREQS     r3, r0, r1, LSR #10 ; r3 = r0 | (r1 / 1024)
```

r0	r1	r2	r3	NZCV
b'100000000'	1024	4	b'100000001'	0000

The memory is shown in binary for r0 and r3 for clarity.

This example has shown most of the different options available to code an instruction and given lots of samples of Data Processing instructions. Until we are actually working towards a goal for a program, it will not be entirely clear how all of these instructions are going to come together to make something useful happen. In fact, Data Processing instructions by themselves probably will not suffice to write a program. We have to get into the memory access instructions to load and store memory to and from RAM, ROM and peripheral registers.



## 6.5 Load and Store Instructions

The ARM is classified as a load-and-store architecture and this functionality is fundamental to its operation. The instruction set allows data processing to occur only on the core's registers, so there must be a way to get data in and out of the core from RAM or ROM. There are four Load – Store instructions that enable you to accomplish that very task. The good news is that you really only need to learn half of them because the other half work essentially the same way but in the opposite direction (i.e. moving from RAM to registers vs. registers to RAM). The bad news is that the operands required to make the instructions work are quite complex, though it starts out very easy in the simplest form.

Before looking at the instructions, let us get the hardest part out of the way: addressing. What should be obvious is that if you want to access a memory location, you have to know where it is. The 32-bit address of the memory location you want then has to be loaded into a register, which requires an immediate to specify the address. Though sometimes the address will be one that can be generated as an immediate, it often is not. If it is not, then you have the same problem of creating the number for the address in one of the core registers so it can then be dereferenced to get the data that is present there. For now, we will assume that the addresses can be generated as an immediate or we can use a few techniques to get close and then manipulate the value slightly to get what the correct value.

Addresses of special function and general purpose registers at certain locations in RAM are loaded in the same way as each other. For special function registers, you are limited to the hard-coded addresses assigned to peripherals of the LPC214x (i.e. the peripheral addresses we looked at in Figure 6.1.2.2). For general purpose RAM, you can use whatever location you want provided it is in the allowable RAM space between 0x4000 0000 and 0x4000 3FFF inclusive. Ideally, the address is one whose immediate value can be generated, in which case a simple move instruction can be used. Assume you want to access memory at the first address in the user RAM space, 0x4000 0000. Can you load this immediate given the criteria defined for immediates? Convert to decimal, then take the log 2 of the number (if your calculator does not have log base 2, then you can do log-base-10 of the number divided by log-base-10 of 2).

$$\begin{aligned} 0x40000000 &\xrightarrow{\text{HexToDec}} 1073741824 \\ \log_{10} 1073741824 / \log_{10} 2 \\ &= 30 \end{aligned}$$

So indeed, this address can be computed with the immediate generator on the ARM.

The first peripheral, the Watchdog timer, is another good example as it lives at 0xE000 0000, which is a number that can be created by  $224 \times 2^{(2 \times 12)}$ . So simply use a MOV instruction:

```
GET_ADDR    MOV    r1, #0xE000 0000 ; r1 = Watchdog timer address
```



If the address does not fall on an available immediate, then you could create it by adding an offset to a base register. For example, adding 0x4000 (yes, a valid immediate) to r1 above would give you 0xE000 0004, which is the next peripheral device. You can also use tricks like pre- and post-increments within some instructions or simple immediate additions and subtractions to the base address. When you start writing in C, take some time to look at the often clever way the compiler will create numbers for you.

### 6.5.1 Single Load and Store Instructions

Let us use an example to see how load and store instructions are used. LDR is “load register from memory” and STR is “store register to memory.” In their simplest forms, they use a register and a pointer to a memory location as arguments. For notation, a pointer is shown as the register in square brackets, like this: [r1]. The example shown here will load the current value in 0x4000 0000, multiply it by 4, then store it back to the next 32-bit location in RAM, 0x4000 0004. Register r1 will hold the address, and r0 will be used during the arithmetic instructions.

Again, the best way to traverse this exercise is to run it in IAR. In the same download as the previous exercise, there is another project with chapter6b.s as the source file that contains the code in the example. Make sure the source files for chapter6.s are closed, then open up chapter6b.s. When setting up the debug environment, make sure you can see a Register window and both the Memory and Symbolic Memory windows. In both memory windows, enter “0x40000000” in the “Go to” areas so that the windows show you the memory addresses we want to look at. Your environment should resemble Figure 6.5.1.1.

The memory starts out like this:

r0	r1	0x4000 0000	0x4000 0004	NZCV
?	?	?	?	????

First, make r1 point to the RAM space by loading the RAM address:

```
MOV    r1, #0x40000000    ; r1 = the target address (this
                           ; is a valid immediate)
```

Note that there is nothing that tells r1 that the value it has been loaded with is a pointer – we just know that is what that number will be used for.



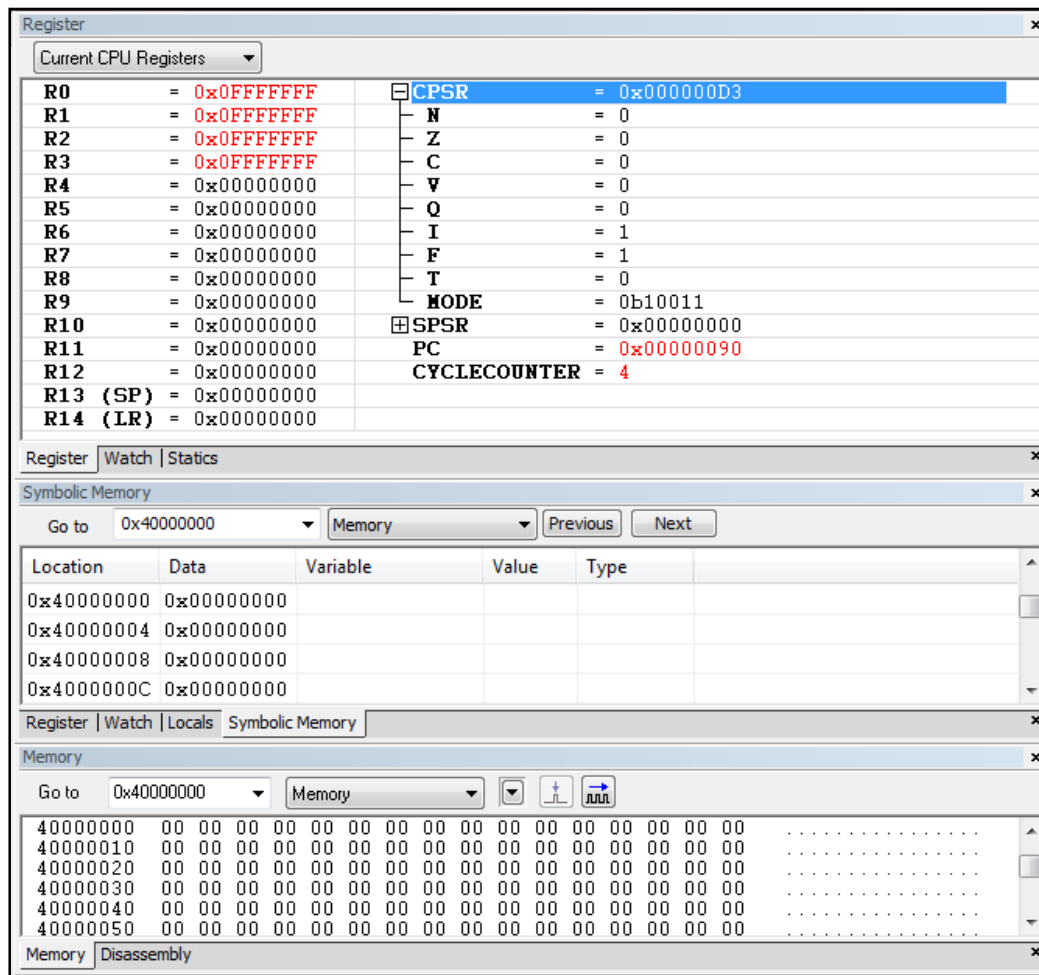


Figure 6.5.1.1: Register, Symbolic Memory and Memory windows for this exercise

We want the number 5000 to be loaded into the RAM location 0x40000000. The first step is to get the number 5000 into a register, so can we do the following?

```
MOV    r0, #5000           ; r0 = 5000 ?
```

No! If you try it, the assembler returns:  
 Error[400]: Expression out of range

The number 5000 is not a valid number for the immediate generator. So how do we get that number? We can use an LDR instruction that will automatically do two things: store the number 5000 in a flash location, and then update the instruction to access that location to get the value into a core register. The location where this number is stored is known as the “literal pool” where a bunch of constants hang out in flash memory (they are put there at assemble time). We will look more closely at that in the next chapter. The syntax is:

```
LDR    r0, =5000           ; r0 = 5000
```



Now we have:

r0	r1	0x4000 0000	0x4000 0004	NZCV
5000	0x4000 0000	?	?	0000

Now dereference the pointer in r1 to store the value from r0 into RAM. The syntax requires square brackets around the r1 mnemonic to indicate you want to use its contents as an address pointer. The typical destination, source operand order is also reversed for a STR instruction:

```
STR      r0, [r1]      ; *r1 = r0
```

r0	r1	0x4000 0000	0x4000 0004	NZCV
5000	0x4000 0000	5000	?	0000

If you look at your memory windows (and have correctly set them to show you address 0x40000000), you should see the new value written (Figure 6.5.1.2 shows what you should be seeing). Notice two things:

1. 0x1388 is hex for 5000 (this is where number conversions start to get handy)
2. The values are in Little Endian

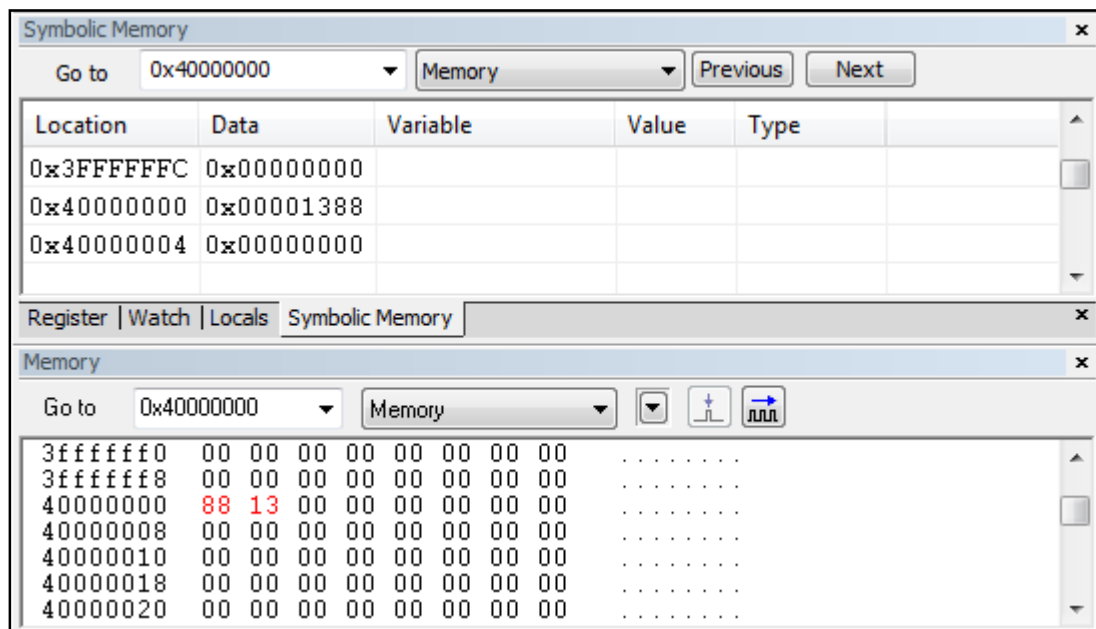


Figure 6.5.1.2: RAM location 0x40000000 after 5000 is written



Now we will do a bunch of other operations to ensure you are comfortable with loading and storing.

Multiply the value in r0 using a shift, update r1 to the address of the second RAM register we want to use, and store the value back by once again dereferencing r1:

```
LSL      r0, r0, #2      ; Multiply r0 by 4
ADD      r1, r1, #4      ; r1 += 4 (new RAM address)
STR      r0, [r1]        ; *r1 = r0
```

r0	r1	0x4000 0000	0x4000 0004	NZCV
20000	0x4000 0004	5000	20000	0000

LDR and STR in their simplest form are indeed easy using the simplest of register-indirect addressing with pointers. For the remainder of this discussion, remember that LDR and STR instructions with more complicated operands work exactly the same.

Using a separate ADD instruction like we did above seems a little wasteful since we know we have at least a few bits in a 32 bit instruction where an offset could be coded or the barrel shifter could be triggered. Indeed, the ARM instruction set supports this. The next easiest form of the LDR instruction is to include an offset in Operand 2. This allows the programmer to load a base address to a particular register, then use the base + offset at anytime without destroying the original base address, which is called pre-indexing. So the last ADD & STR instruction combination could simply be replaced by:

```
STR      r0, [r1, #4]    ; *(r1 + 4) = r0
```

If you are following along in the code in IAR, executing the above line writes 20000 to the next RAM location, 0x40000008. Now we can introduce the first flag to this instruction that modifies its behavior further. In this case, the flag is written in the operand fields and is an exclamation mark (!). This tells the processor to not only increment the address pointer, but also to store the incremented value as the new base address in the register that was used for the pointer. This is called auto-indexing since the base register is automatically updated. This is very useful inside loops for indexing consecutive data.

```
STR      r0, [r1, #4]!   ; *(r1 + 4) = r0
                          ; r1 += 4
```

A slight variation of that is to use post-indexing. In this mode, the base address is dereferenced and then incremented. No exclamation mark is used since the operands have a unique form already.

```
STR      r0, [r1], #4    ; *r1 = r0
                          ; r1 += 4
```



In both of the last two instructions, you should not see any RAM values change because of what the value of r1 is and when the address increments. Study this carefully in the simulator to ensure you really understand why!

In any of the above three examples, the immediate value can be replaced by an offset held in another register. Furthermore, said value can be shifted (the argument format is almost identical to that already discussed in detail for “Op2” in Data Processing instructions, except you can use a positive or negative offset register). The last variations of the LDR and STR instructions come with the [size] flag that can be added. It is possible to move bytes and half words by adding either “B” or “H” to the instruction mnemonic before the condition code. For example, consider this memory (do not forget that ARM uses little-endian storage):

r0	r1	r2	0x4000 0004	NZCV
?	0x4000 0000	8	0x1234 FEDC	0000

Load only a half word of data from RAM to r0, then check if r0 and r2 are the same:

```
LDRH      r0, [r1, #4]          ; r0 = The half word starting
                                ; at *(r1 + 4)
TST       r0, r2                ; Test if equal
```

r0	r1	r2	0x4000 0010	NZCV
0x0000FEDC	0x4000 0000	8	?	0000

The two high bytes in r0 get filled with zeros. The memory map as shown above is updated for the upcoming destination register which will be 0x4000 0010. The starting word address for this is 16 bytes away from the address currently in r1. For fun, we will only execute the next instruction if r0 and r2 are not equal based on the test that was done. After the instruction, we want r1 to point to the destination register. All this is done in a single instruction:

```
STRNE     r0, [r1, +r2, LSL #1]! ; *(r1 + r2 x 2) = r0
                                ; r1 += r2 x 2
```

r0	r1	r2	0x4000 0010	NZCV
0x0000FEDC	0x4000 0010	8	0x0000FEDC	0000

The point of the last instruction is to demonstrate how many things can be accomplished with only a single instruction on an ARM processor. The same functionality could take as many as 10



instructions on other processors, meaning 10x the memory resources and 10x the clock cycles and 10x the power consumption. This shows how powerful the instruction set is, and thus why ARM is such a great processor core.

## 6.5.2 Multiple Load and Store

In addition to the two instructions that allow single-word (or byte) memory access, there is also a load and corresponding store instruction that can access a range of words. These instructions allow up to 16 registers to be transferred! Though this does not happen in a single instruction cycle (it takes about 2 cycles per register), it is invoked with a single instruction so there is potentially a great savings in code space. Note that the multiple load/stores must take place on 32-bit words only.

The root of these instructions is as follows:

```
LDM          ; Load multiple
STM          ; Store multiple
```

The instruction operands work just like the single-word instructions where LDM follows the form “destination, source” and STM is “source, destination.” Transferring multiple words implies a pointer must be moving through memory adding or subtracting offsets from a base register and indeed this is the case. So to the root of the instruction, a two-letter prefix has to be added. The prefix tells the base register to increment after each load (IA), increment before each load (IB), decrement after each load (DA), or decrement before each load (DB). Load/store multiples are also conditionally executed like all the other ARM instructions with one of the post-fixed condition codes that you have already seen.

The operands for the instructions are a pointer to the base address for the operation, and a register list with up to 16 registers specified (i.e. r0 thru r15, inclusive). The list is provided in braces {} and is not order-dependant. It can have individual registers specified and comma-delimited, or use ranges with a “-” or any combination. The instruction will always load/store the lowest register first, then the next highest, next highest after that and so on. For example, the following lists are identically executed:

- {r15, r2-r7, r1, r14, r12}
- {r1-r7, r12, r14, r15}
- {r1, r2, r3, r4, r5, r6, r7, r12, r14, r15}
- {r1-r7, r12, r14, pc}

Note the very important fact: r14 (the link register) and r15 will always be loaded last in a list of registers. Clearly, writing values to these registers is significant as they are the return address and program counter. This provides a very efficient way to enter and exit function calls, where a list of variables are stored (to a stack) along with the return address and program counter, the function executes, and then the values are restored.



Since the load / store multiples can be used for stack-type manipulation, there is an alternate form of the prefixes that can be used. This helps the programmer when managing a stack in assembly, even though the instruction behaves the same way. There are four forms of stack that can be supported:

- Full ascending (FA): the stack grows upwards in memory and the stack pointer points at the highest address of valid data. So the next item on the stack must first increment the stack pointer, then write the value. This corresponds to IB when storing (pushing) to the stack, and DB when loading from (popping) the stack.
- Empty ascending (EA): the stack grows upwards in memory and the stack pointer points at the next free memory word. To store (push) to the stack, the pointer is dereferenced and then incremented (IA). When loading from (popping) the stack, the stack pointer must first be decremented, then the value read (DB)
- Full descending (FD): these stacks grow down in memory and the stack pointer always points to the lowest address of a valid item. Storing / push requires the stack pointer to be decremented first, then the new value is written (DB). Loading / popping reads the value, then increments (IA)
- Empty descending (ED): the stack grows down in memory and the stack pointer points at the next location where data will be added. Storing / pushing takes place by adding data to the stack pointer location then decrementing (DA), while loading / popping must move the stack pointer up, then load the value (IB).

Figure 6.5.2.1 summarizes the common relationship between the two optically different approaches to using prefixes on the load and store multiply instructions.

Load Multiple (LDM)	
Data Perspective	Stack Perspective
LDMIA (increment after)	LDMFD (full descending)
LDMIB (increment before)	LDMED (empty descending)
LDMDA (decrement after)	LDMFA (full ascending)
LDMDB (decrement before)	LDMEA (empty ascending)

Store Multiple (STM)	
Data Perspective	Stack Perspective
STMIA (increment after)	STMEA (empty ascending)
STMIB (increment before)	STMFA (full ascending)
STMDA (decrement after)	STMED (empty descending)
STMDB (decrement before)	STMFD (full descending)

*Figure 6.5.2.1: LDM and STM prefix equivalencies*



It is up to you as a user to decide what perspective you will choose to implement if you find yourself coding with multiple load and stores. In general, the Data Perspective is easier to understand, so at least for practicing with these types of instructions you should stick with that.

The last augmentation to the multiple load / store instructions are special characters that can change the instruction behavior. Adding an exclamation point to the pointer register will cause it to be updated with whatever the last base address of the load was. For example, if you started with a pointer to memory 0x4000 0000 and loaded 4 registers with LDMIA with the ! on the base operand, the pointer register would end up at 0x4000 0000 + 16 = 0x4000 0010. There is also a “^” character that can be used in the instruction, but this will not be covered until we get into interrupts later on.

Here are some examples of the load / store multiple instructions in action. For the sake of the example, we will show only the last three bytes of the RAM addresses 0x40000400 thru 0x40000500. The chapter 6 sample code project with chapter6c.s should be used here. Set up your debugging view to have two Memory windows so you can see addresses starting at 0x40000400 and 0x40000500, and a Watch window with decimal register values as shown in Figure 6.5.2.2.

The starting values in memory look like this (four character text strings can be stored in one 32-bit address since each character is only 8 bits):

r0	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	r11
0x400	0x500	?	?	?	?	?	?	?	?	?	?
0x400	0x404	0x408	0x40C	0x410	0x414	0x418	0x41C	0x420	0x424	0x428	0x42C
100	110	115	157	198	331	134	642	EXAM	PLE_	TEXT	BLAH
0x500	0x504	0x508	0x50C	0x510	0x514	0x518	0x51C	0x520	0x524	0x528	0x52C
?	?	?	?	?	?	?	?	?	?	?	?

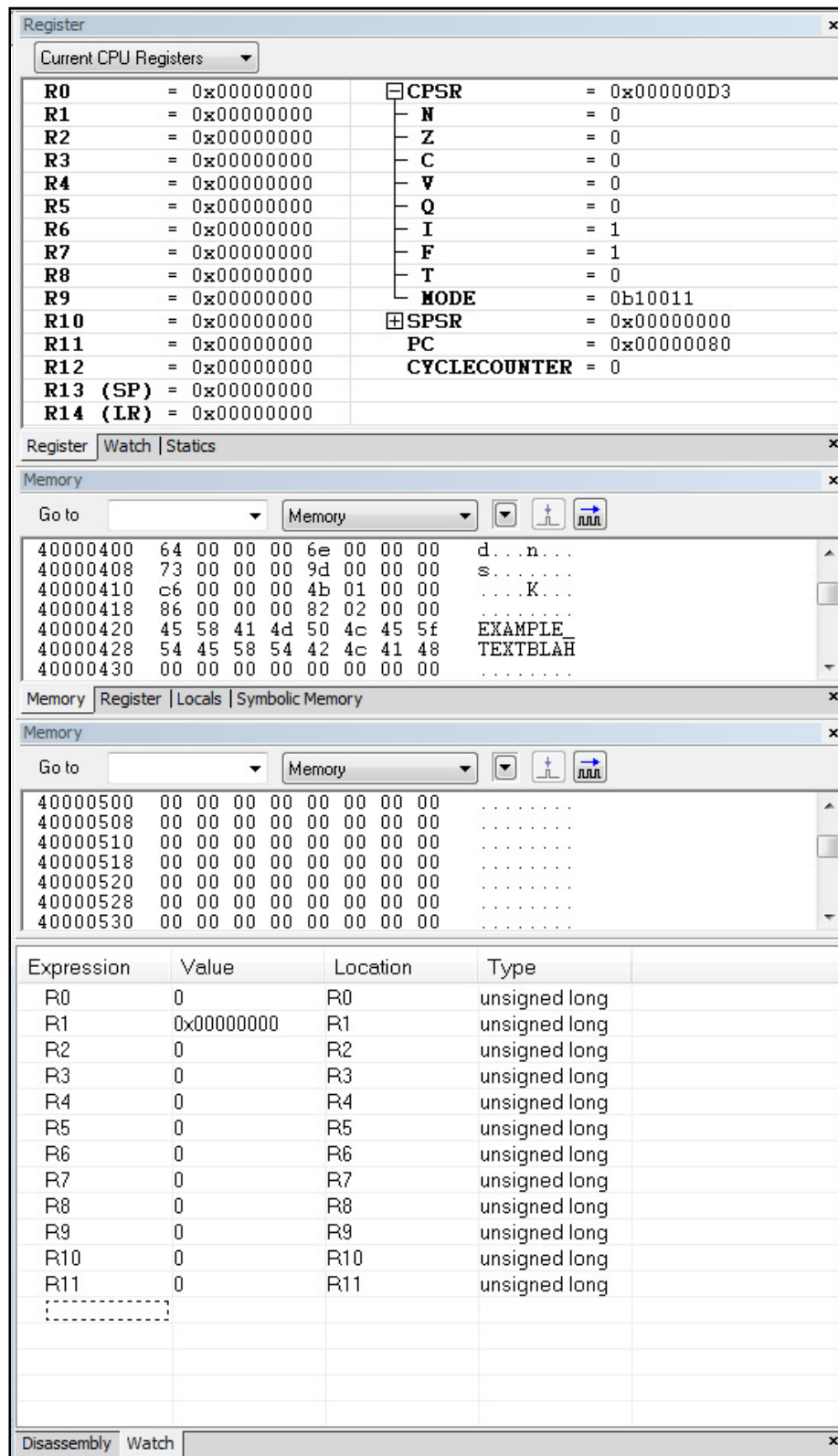


Figure 6.5.2.2: Debugger setup for this exercise





First we will load three consecutive registers with data starting at the location that r0 points to:

```
LDMIA    r0, {r2-r4}    ;
```

The only memory that changes are the registers r2, r3 and r4 which are loaded with the values from 0x400, 0x404, and 0x408, respectively. The postfix IA in the LDR instruction means that the pointer r0 increments after each load, but the incremented value is not written back to r0.

r0	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	r11
0x400	0x500	100	110	115	?	?	?	?	?	?	?

Now load some more registers that are not consecutive and update r0 with the end pointer result by adding “!” to the first operand. “IB” means the value in r0 and thus in the index to memory will increment before the value is read.

```
LDMIB    r0!, {r10, r6, r9}    ;
```

Remember that the order of the registers in the register list is not considered, so r6 gets loaded first, then r9, then r10. The assembler will warn you about this, but allow it. Writing the code like this is crappy programming because it is unclear! The instruction also increments before the value is retrieved, so the memory locations that are accessed are 0x404, 0x408 and 0x40C.

r0	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	r11
0x40C	0x500	100	110	115	?	110	?	?	115	157	?

For fun, move the r0 pointer to the last location where data is stored ( $0x40C + 0x20 = 0x42C$ ) and fill up the remaining registers but this time use a descending postfix on the LDM instruction as if this was a FILO stack. Update the pointer after complete.

```
ADD      r0, r0, #0x20    ;  
LDMDB    r0!, {r11, r8, r7, r5}    ;
```

In writing the register list, it is implied that r11 will be loaded with the value at 0x42C as that is the first word to be retrieved, which is in fact the case here though that does not have anything to do with the order of the list. The LDM instruction always loads the registers with the lowest-numbered register from the lowest memory address (start\_address), through to the highest-numbered register from the highest memory address (end\_address). So the result of the instruction looks like this:



r0	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	r11
0x41C	0x500	100	110	115	EXAM	110	PLE_	TEXT	115	157	BLAH

Now store all the updated registers to the memory that r1 points to. In this case, the entire list is desired and we will use the prefix as if we were writing to an ascending empty stack. Note that stack writes would certainly want to update the stack pointer, in this case r1, so an '!' is included.

```
STMEA    r1!, {r2-r11} ;
```

r0	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	r11
0x41C	0x528	100	110	115	EXAM	110	PLE_	TEXT	115	157	BLAH
0x400	0x404	0x408	0x40C	0x410	0x414	0x418	0x41C	0x420	0x424	0x428	0x42C
100	110	115	157	198	331	134	642	EXAM	PLE_	TEXT	BLAH
0x500	0x504	0x508	0x50C	0x510	0x514	0x518	0x51C	0x520	0x524	0x528	0x52C
100	110	115	EXAM	110	PLE_	TEXT	115	157	BLAH	?	?

Actually implementing the load / store multiples turns out to be quite straight forward. Life in this world will get a bit more complicated once you start using these types of instructions for managing stacks and jumping around functions and exceptions, but you have a good foundation on which to build on. It is also likely that a lot of that kind of detail will be left up to a compiler to handle since you will probably be coding anything but simple functions in assembly (unless of course you are forced to do it as part of this course!).

## 6.6 Branch Instructions

We know that code is executed from flash as the program counter moves through the memory and addresses each instruction by incrementing address. If left entirely on its own, the program counter always starts at address 0x0000 on power-up and will increment by 4 bytes (1 word) after each fetch. Clearly there must be a mechanism for changing the program counter by a different amount to enable looping, gotos and function calls -- there certainly is.

There are three different branch instructions:

1. B: Branch without the intent of returning. This acts like a goto-type instruction. The only thing that occurs is the program counter is loaded with the target address, which



will cause the program to “jump” to the new address. Since there is no record of where you came from, you cannot possibly get back.

2. BL: Branch and link. This is a function call. The program counter is loaded to the address of the function you want to call, and the link register, r14 is loaded with the return address. In other words, the address of the instruction in memory that immediately follows the BL is saved in r14 so you can return from where you came and resume code execution.
3. BX: Branch and exchange: this instruction branches to code by changing the program counter and also sets the Thumb flag in the CPSR to switch the processor into 16-bit Thumb mode. Since we do not bother to look at Thumb mode on this processor, we will not explore this further.

The simple Branch is typically used for program flow as you are writing code where instructions need to be skipped or loops need to be implemented. Therefore, the B instruction is very often combined with a condition code to make the branch occur only if certain flags are in a particular state. The classic example is a for loop which will be explored in the next chapter.

Calling functions is something all but the simplest programs must do. The ARM accomplishes this with a branch instruction that loads the program counter with the function’s address, but it also invokes logic to grab the return address and store it in r14 (the link register). An example is shown in the code snippet here, with sample addresses as labels:

```
0x100      BL      function_x      ; Call the function x: loads
                                           ; pc with 0x200 and LR with 0x104
0x104      ...
                                           ; Code continues
0x200      ...
                                           ;

function_x  ...                        ; Do some stuff in the function
           Need to return now!
```

Instead of an explicit return instruction as you will find in some processors, ARM functions are exited with a move that simply takes the address stored in r14 and loads it back to the program counter. The format always looks like this:

```
MOV        pc, r14                    ; Return from function / subroutine
```

The somewhat obvious issue is that there is only one link register thus you must do some saving and restoring if calling and returning from functions greater than one deep. It is almost guaranteed that any program you write will need some sort of stack structure where working registers are saved, so it is logical to use the stack to store return addresses. So assuming you are implementing an empty ascending stack (r13 is the stack pointer), the function code will look something like this:



```
0x100      BL      function_x      ; Call the function x: loads
                                           ; pc with 0x200 and LR with 0x104
0x104      ...
                                           ; Code continues
0x108      ...
                                           ;

0x200

function_x  ...      ; Do some stuff in the function
0x20C      STR      r14, [r13], #4  ; Store the current r14 to the
                                           ; stack pointer, then increment the
                                           ; stack pointer

0x210      BL      function_x2 ; Call the next function,
                                           ; r14 = 0x214 and pc = 0x300
0x214      ...
                                           ; Do more stuff
           LDR      pc, [r13, #-4]! ; Decrement stack pointer then read
                                           ; off the address in to program
                                           ; counter.

0x300
function_x2 ...      ; Do some stuff
           MOV      pc, r14          ; Return to function_x
                                           ; pc = 0x214
```

If you are also passing parameters or saving registers via the stack, load / store multiple instructions can be used as well. Always be careful to be aware of the order of the different values on the stack. Stacks are always first-in-last-out (FILO). At this level of complexity, you are begging for errors unless you spend a lot of time developing a careful system that works. This is one of the things that a C compiler has to figure out and in most cases you can take it for granted that the compiler is going to get it right.

## 6.7 Code design

One of the hardest parts about writing in assembly is designing code. A good programmer in any language will spend a lot of time planning a program. This often means breaking down a problem into manageable parts and very carefully considering how those parts will work together. Once a problem is sufficiently modularized, designing each module is critical to ensure it is efficient and bug-free. There are always many different ways to solve a problem and even though assembly language programming is low-level with only a relatively small number of choices for instructions, you will have lots of choices on how to write a chunk of code. Some will run faster, some will be bloated but perhaps more safe, some will be crappy and probably have to be rewritten.

Knowing the instruction set and how to write good assembly code is extremely important for embedded development even if you write all your programs in C. By knowing the capabilities of the instruction set and how bits move around in the processor core, you can write programs to take advantage of the built-in power of the instructions.

For a very simple example, consider an analog to digital (A to D or A/D) conversion of a slightly noisy DC signal. You want to take some readings and use the average of those readings as the



final result. If you were doing it in your head, you would probably choose 10 samples since division by 10 is easy on paper. However, when an embedded system is averaging numbers, dividing by a power of 2 is far more efficient since it can be done with simple bit-shifting rather than a division routine. You can even gain small efficiencies by doing things like using count-down loops instead of count-up loops. If you count up like in a for loop (for j = 0; j < 32, j++) you have to load a limit register with 32, increment your counter, then do a subtraction to see if you have reached to limit value. If you count down instead, you save a few instructions since you only have to load the counter once, then decrement it until it reaches zero (you save the extra subtraction of the current value against the limit value). It might not seem like much, but actually is a 33% improvement in size and speed which is significant.

There has been a lot of information presented in this chapter, and chances are that it is entirely new to you unless you have programmed in assembly language before. It is important to review this chapter and become familiar with the instruction set summary provided. This tends to be a bit more difficult to comprehend when it is all written down on paper, but in no time you should be able to grasp the concepts that are actually very simple – they are just new!

## **APPENDIX A: ARM7TDMI Instruction Set Summary**

---



### ARM7TDMI-S Instruction Set

Mnemonic	Type	Syntax	Action	Flags	Description	Cycles
ADC	Data Processing	ADC[cond][s] Rd, Rn, <Op2>	Rd = Rn + Operand2 + Carry	N Z C V	Add with carry	
ADD	Data Processing	ADD[cond][s] Rd, Rn, <Op2>	Rd = Rn + Op2	N Z C V	Add	
ADR	Data Processing	ADR[cond] Rd, <label>	Rd = <label>		Form PC-relative address	
AND	Data Processing	AND[cond][s] Rd, Rn, <Op2>	Rd = Rn AND Operand2	N Z C	Bitwise logical AND	
ASR	Data Processing	ASR[cond][s] Rd, Rm, <Rs sh>	Rd = ASR(Rm, Rs sh)	N Z C	Arithmetic shift right	
B	Branch	B <label>	R15 = address		Branch (like goto)	
BIC	Data Processing	BIC[cond][s] Rd, Rn, <Op2>	Rd = Rn AND NOT Operand2	N Z C	Bit clear	
BL	Branch	BL <label>	R15 = address; R14 = return address		Branch with link (function call)	
BX	Branch	BX Rm	R15 = Rm; Target is ARM if Rm[0] is 0		Branch and Exchange	
CMN	Data Processing	CMN[cond] Rn, <Op2>	Update CPSR flags on Rn + Operand2	N Z C V	Compare Negative	
CMP	Data Processing	CMP[cond] Rn, <Op2>	Update CPSR flags on Rn - Operand2	N Z C V	Compare	
EOR	Data Processing	EOR[cond][s] Rd, Rn, <Op2>	Rd = Rn EOR Operand2	N Z C	Bitwise logical Exclusive OR	
LDM	Load-Store	LDM[IA IB DA DB][cond] Rn[!], <reglist-PC> LDM[IA IB DA DB][cond] Rn[!], <reglist+PC> LDM[IA IB DA DB][cond] Rn[!], <reglist+PC>^ LDM[IA IB DA DB][cond] Rn, <reglist-PC>^	Load list of registers from Rn Load registers, PC = [address][31:1] Load registers, branch, CPSR = SPSPR (Exception) Load list of User mode registers from Rn (Privileged)		Load multiple registers	
LDR	Load-Store	LDR[size][T][cond] Rd, Rn [, #offset][!] LDR[size][T][cond] Rd, [Rn], #offset LDR[size][cond] Rd, [Rn, +/-Rm [, opsh][!] LDR[size][T][cond] Rd, [Rn], +/-Rm [, opsh] LDR[size][cond] Rd, <label>	Rd = [address, size] Rd = [address, size] Rd = [address, size] Rd = [address, size] Rd = [label, size]		Load register from memory (Immediate offset) Post-indexed, immediate Register offset Post-indexed, register PC-relative	
LSL	Data Processing	LSL[cond][s] Rd, Rm, <Rs sh>	Rd = LSL(Rm, Rs sh)	N Z C	Logical shift left	
LSR	Data Processing	LSR[cond][s] Rd, Rm, <Rs sh>	Rd = LSR(Rm, Rs sh)	N Z C	Logical shift right	
MLA	Data Processing	MLA[cond][s] Rd, Rm, Rs, Rn	Rd = (Rn + (Rm * Rs))[31:0]	N Z C	Multiply Accumulate	
MOV	Data Processing	MOV[cond][s] Rd, <Op2>	Rd = Operand2	N Z C	Move register or constant	
MRS	Status Register	MRS[cond] Rd, <PSR>	Rd = PSR		Move PSR status/flags to register	
MSR	Status Register	MSR[cond] <PSR> <fields>, Rm MSR[cond] <PSR> <fields>, #, <imm8m>	PSR = Rm (selected bytes only) PSR = imm8m (selected bytes only)		Move register to PSR status/flags	
MUL	Data Processing	MUL[cond][s] Rd, Rm, Rs	Rd = (Rm * Rs)[31:0]	N Z C	Multiply	
MVN	Data Processing	MVN[cond][s] Rd, <Op2>	Rd = 0xFFFFFFFF EOR Operand2	N Z C	Move negative register	
ORR	Data Processing	ORR[cond][s] Rd, Rn, <Op2>	Rd = Rn OR Operand2	N Z C	Bitwise logical OR	
POP	Data Processing	POP <reglist>	Same as LDM SP!, <reglist>		Pseudo instruction for popping the stack	
PUSH	Data Processing	PUSH <reglist>	Same as STMDB SP!, <reglist>		Pseudo instruction for pushing stack	
ROR	Data Processing	ROR[cond][s] Rd, Rm, <Rs sh>	Rd = ROR(Rm, Rs sh)	N Z C	Rotate right	
RRX	Data Processing	RRX[cond][s] Rd, Rm	Rd = RRX(Rm)	N Z C	Rotate right with extend	
RSB	Data Processing	RSB[cond][s] Rd, Rn, <Op2>	Rd = Operand2 - Rn	N Z C V	Reverse subtract	
RSC	Data Processing	RSC[cond][s] Rd, Rn, <Op2>	Rd = Operand2 - Rn - NOT(Carry)	N Z C V	Reverse subtract with carry	
SBC	Data Processing	SBC[cond][s] Rd, Rn, <Op2>	Rd = Rn - Operand2 - NOT(Carry)	N Z C V	Subtract with carry	
SMLAL	Data Processing	SMLAL[cond][s] RdLo, RdHi, Rm, Rs	RdHi, RdLo = signed(RdHi, RdLo + Rm * Rs)	N Z C V	Signed multiply and accumulate long	
SMULL	Data Processing	SMULL[cond][s] RdLo, RdHi, Rm, Rs	RdHi, RdLo = signed(Rm * Rs)	N Z C V	Signed multiply long	
STM	Load-Store	STM[IA IB DA DB][cond] Rn[!], <reglist> STM[IA IB DA DB][cond] Rn[!], <reglist>^	Store list of registers to [Rn] Store list of User mode registers to [Rn] (Privileged)		Store multiple	
STR	Load-Store	STR[size][T][cond] Rd, Rn [, #offset][!] STR[size][T][cond] Rd, [Rn], #offset STR[size][cond] Rd, [Rn, +/-Rm [, opsh][!] STR[size][T][cond] Rd, [Rn], +/-Rm [, opsh]	[address, size] = Rd [address, size] = Rd [address, size] = Rd [address, size] = Rd		Store register to memory (Immediate offset) Post-indexed, immediate Register offset Post-indexed, register	
SUB	Data Processing	SUB[cond][s] Rd, Rn, <Op2>	Rd = Rn - Operand2	N Z C V	Subtract	
SWI	Software Interrupt	SWI <imm24>	Software exception		Software interrupt	
SWP	Data Processing	SWP Rd, Rm, [Rn]	temp = [Rn]; [Rn] = Rm; Rd = temp		Swap register with memory	
TEQ	Data Processing	TEQ Rn, <Op2>	Update CPSR flags on Rn EOR Operand2	N Z C	Test bits	
TST	Data Processing	TST Rn, <Op2>	Update CPSR flags on Rn AND Operand2	N Z C	Test bitwise equality	
UMLAL	Data Processing	UMLAL[cond][s] RdLo, RdHi, Rm, Rs	RdHi, RdLo = unsigned(RdHi, RdLo + Rm * Rs)	N Z C V	Multiply unsigned accumulate long	
UMULL	Data Processing	UMULL[cond][s] RdLo, RdHi, Rm, Rs	RdHi, RdLo = unsigned(Rm * Rs)	N Z C V	Multiply unsigned long	



### Notes on instruction arguments

!	Flag to update base register after data transfer (pre-indexed)
[]	Indicates field that is optionally present, otherwise blank
+/-	Plus or minus (+ may be omitted for default positive)
<>	Indicates field that must be present
ASR	Arithmetic shift right (only bits 0 – 30, preserving sign bit)
cond	An optional condition code that, if present, will evaluate first to determine if the instruction is executed (see table for allowed conditions)
DA	Decrement after
DB	Decrement before
fields	c, f, s, x (Control field mask PSR[7:0]; Flags field mask PSR[31:24]; Status field mask PSR[23:16]; Extension field mask PSR[15:8])
IA	Increment after
IB	Increment before
imm8m	32-bit constant formed by right-rotating and 8-bit value by an even number of bits
label	Mnemonic for a code address (for PC-relative arguments, this must be within +/- 4092 of current PC address)
LSL	Logical Shift Left
LSR	Logical Shift Right
offset	For size=B, any value -4095 to +4095; for size = SB, H, or SH, any value -255 to +255
Op2	Operand 2 (see table for allowed arguments)
opsh	Optional shift
PSR	Either CPSR or SPSR
Rd	Destination register
reglist	A comma-separated list of registers enclosed in braces {}
reglist+PC	A reglist including the PC
reglist-PC	A reglist that must not include the PC
Rm	Any generic register used in the instruction operands
Rn	Operand 1 register
ROR	Rotate Right (LSB shifts into MSB)
RRX	Rotate Right through Carry (LSB shifts into carry, carry shifts to MSB)
Rs	Any generic register (intended to hold the shift or multiply amount)
Rs sh	Either Rs or an immediate shift value
s	Flag to cause instruction result to update CPSR flags
size	B, SB, H or SH (Byte, Signed Byte, Halfword, Signed Halfword). Note SB and SH are not allowed for STR instructions
T	Flag to set User mode privilege

Condition Fields			Operand 2		Other notes
Mnemonic	CPSR Flags	Description			
EQ	Z set	Equal	Immediate value	#<imm8m>	
NE	Z clear	Not equal	Register, optionally shifted by constant	Rm {, <opsh>}	
CS / HS	C set	Carry set / Unsigned higher or same (>=)	Register, logical shift left by register	Rm, LSL Rs	
CC / LO	C clear	Carry clear / Unsigned lower (Less than)	Register, logical shift right by register	Rm, LSR Rs	
MI	N set	Negative (Less than)	Register, arithmetic shift right by register	Rm, ASR Rs	
PL	N clear	Positive or zero	Register, rotate right by register	Rm ,ROR Rs	
VS	V set	Overflow	Optional shifts by constant (opsh): No shift: Rm (same as Rm, LSL #0) Logical shift left: Rm, LSL #<shift> where allowed shifts are 0-31 Logical shift right: Rm, LSR #<shift> where allowed shifts are 1-32 Arithmetic shift right: Rm, ASR#<shift> where allowed shifts are 1-32 Rotate right: Rm, ROR #<shift> where allowed shifts are 1-31 Rotate right with extend: Rm, RRX		
VC	V clear	No overflow			
HI	C set and Z clear	Unsigned higher			
LS	C clear or Z set	Unsigned lower or same			
GE	N equals V	Signed greater than or equal			
LT	N not equal V	Signed less than			
GT	Z clear, N = V	Signed greater than			
LE	Z set or N not = V	Signed less than or equal			
AL	Any	Always (normally omitted)			





## **APPENDIX B: Assembler Information**

---



Number representations in the Assembler. Note that only decimal and hexadecimal representations are available in C files.

Integer type	Example
Binary	1010b, b'1010
Octal	1234q, q'1234
Decimal	1234, -1, d'1234
Hexadecimal	0FFFFh, 0xFFFF, h'FFFF

ASCII constants can consist of any number of characters enclosed in single or double quotes. Only printable characters and spaces may be used in ASCII strings. If the quote character itself is to be accessed, two consecutive quotes must be used:

Format	Value
'ABCD'	ABCD (four characters).
"ABCD"	ABCD '\0' (five characters the last ASCII null).
'A' 'B'	A 'B
'A' ''	A '
' ' ' ' (4 quotes)	'
' ' (2 quotes)	Empty string (no value).
" " (2 double quotes)	Empty string (an ASCII null character).
\ '	', for quote within a string, as in 'I\'d love to'
\\	\, for \ within a string
\ "	", for double quote within a string

Use the symbol control directives to control how symbols are shared between modules. For example, use the PUBLIC directive to make one or more symbols available to other modules. The EXTERN directive is used for importing an untyped external symbol.

If you need to refer to the program location counter in your assembler source code you can use the . (period) sign. For example:

```
SECTION MYCODE : CODE (2)
CODE32
B . ; Loop forever
END
```



### Directive Description Section

\$	Includes a file. Assembler control
#define	Assigns a value to a label. C-style preprocessor
#else	Assembles instructions if a condition is false. C-style preprocessor
#endif	Ends a #if, #ifdef, or #ifndef block. C-style preprocessor
#if	Assembles instructions if a condition is true. C-style preprocessor
#ifdef	Assembles instructions if a symbol is defined. C-style preprocessor
#ifndef	Assembles instructions if a symbol is undefined. C-style preprocessor
#include	Includes a file. C-style preprocessor
#message	Generates a message on standard output. C-style preprocessor
#pragma	Recognized but ignored. C-style preprocessor
#undef	Undefines a label. C-style preprocessor
/*comment*/	C-style comment delimiter. Assembler control
//	C++ style comment delimiter. Assembler control
ARM	Interprets subsequent instructions as 32-bit (ARM) instructions.
CODE16	Interprets subsequent instructions as 16-bit (Thumb) instructions.
CODE32	Interprets subsequent instructions as 32-bit (ARM) instructions.
DATA	Defines an area of data within a code section. Mode control
ELSE	Assembles instructions if a condition is false. Conditional assembly
ELSEIF	Specifies a new condition in an IF...ENDIF block. Conditional assembly
END	Terminates the assembly of the last module in a file. Module control
ENDIF	Ends an IF block. Conditional assembly
EQU	Assigns a permanent value local to a module.
EXTERN	Imports an external symbol.
EXTRN	Imports an external symbol.
IF	Assembles instructions if a condition is true.
IMPORT	Imports an external symbol.
INCLUDE	Includes a file.
LIBRARY	Begins a module; an alias for PROGRAM and NAME.
NAME	Begins a program module. Module control
PUBLIC	Exports symbols to other modules. Symbol control
RADIX	Sets the default base. Assembler control
SECTION	Begins a section. Section control
SECTION_TYPE	Sets ELF type and flags for a section. Section control
THUMB	Interprets subsequent instructions as Thumb execution-mode instructions.
VAR	Assigns a temporary value. Value assignment



Directive	Alias	Description
DC8	DCB	Generates 8-bit constants, including strings.
DC16	DCW	Generates 16-bit constants.
DC24		Generates 24-bit constants.
DC32	DCD	Generates 32-bit constants.
DF32		Generates 32-bit floating-point constants.
DF64		Generates 64-bit floating-point constants.
DS8	DS	Allocates space for 8-bit integers.
DS16		Allocates space for 16-bit integers.
DS24		Allocates space for 24-bit integers.
DS32		Allocates space for 32-bit integers.

## REGISTER SYMBOLS

The following table shows the existing predefined register symbols:

Name	Size	Description
CPSR	32 bits	Current program status register
D0–D15	64 bits	Vector floating-point coprocessor registers for double precision
FPEXC	32 bits	Vector floating-point coprocessor, exception register
FPSCR	32 bits	Vector floating-point coprocessor, status and control register
FPSID	32 bits	Vector floating-point coprocessor, system ID register
R0–R12	32 bits	General purpose registers
R13 (SP)	32 bits	Stack pointer
R14 (LR)	32 bits	Link register
R15 (PC)	32 bits	Program counter
S0–S31	32 bits	Vector floating-point coprocessor registers for single precision
SPSR	32 bits	Saved process status register



## **UNARY OPERATORS – I**

<code>+</code>	Unary plus.
<code>-</code>	Unary minus.
<code>!</code>	Logical NOT.
<code>~</code>	Bitwise NOT.
<code>LOW</code>	Low byte.
<code>HIGH</code>	High byte.
<code>BYTE1</code>	First byte.
<code>BYTE2</code>	Second byte.
<code>BYTE3</code>	Third byte.
<code>BYTE4</code>	Fourth byte.
<code>LWRD</code>	Low word.
<code>HWRD</code>	High word.
<code>DATE</code>	Current time/date.
<code>SFB</code>	Section begin.
<code>SFE</code>	Section end.

**SECTION** *section* :*type* [*flag*] [(*align*)]

*align*    The power of two to which the address should be aligned, in most cases in the range 0 to 30.

The default align value is 0, except for code sections where the default is 1. Note: it appears the default is actually 2, meaning 2<sup>2</sup> or align to 4 bytes which makes sense on a 32 bit processor!

*Type*    The memory type, which can be either CODE, CONST, or DATA.

### **Defining a permanent local value**

Use EQU or = to assign a value to a symbol.

Use EQU or = to create a local symbol that denotes a number or offset. The symbol is only valid in the module in which it was defined, but can be made available to other modules with a PUBLIC directive (but not with a PUBWEAK directive). Use EXTERN to import symbols from other modules.



### **Defining a permanent global value**

Use DEFINE to define symbols that should be known to the module containing the directive. After the DEFINE directive, the symbol is known. A symbol which has been given a value with DEFINE can be made available to modules in other files with the PUBLIC directive. Symbols defined with DEFINE cannot be redefined within the same file.

### **Further Information**

Full IAR assembler documentation is available directly from the IAR IDE in the Help menu. Look for the ARM IAR Assembler Reference Guide and be prepared to learn more about an assembler than you will ever want to know!