Introducing ARM assembly language

by Carl Burch, Hendrix College, October 2011

Contents

In this document, we study *assembly language*, the system for expressing the individual instructions that a computer should perform.

1. Background

We are actually concerned with two types of languages, *assembly languages* and *machine languages*.

1.1. Definitions

A **machine language** encodes instructions as sequences of 0's and 1's; this binary encoding is what the computer's processor is built to execute. Writing programs using this encoding is unwieldy for human programmers, though. Thus, when programmers want to dictate the precise instructions that the computer is to perform, they use an **assembly language**, which allows instructions to be written in textual form. An **assembler** translates a file containing assembly language code into the corresponding machine language.

Let's look at a simple example for ARM's design. Here is a machine language instruction:

1110 0001 1010 0000 0011 0000 0000 1001

When the processor is told to execute that binary sequence, it copies the value from “register 9” into “register 3.” But as a programmer, you'd hardly want to read a long binary sequence and make sense of it. Instead, a programmer would prefer programming in assembly language, where we would express this using the following line.

```
MOV R3, R9
```

Then the programmer would use an assembler to translate this into the binary encoding that the computer actually executes.

But there is not just one machine language: A different machine language is designed for each line of processors, designed with an eye to provide a powerful set of fast instructions while allowing a relatively simple circuit to be built. Often processors are designed to be compatible with a previous processor, so it follows the same machine language design. For example, Intel's line of processors (including 80386, Pentium, and Core i7) support similar machine languages. But ARM processors support an entirely different machine language. The design of the machine language encoding is called the **instruction set architecture** (**ISA**).

And for each machine language, there must be a different assembly language, since the assembly language must correspond to an entirely different set of machine language instructions.

1.2. ISA varieties

Of the many ISAs (instruction set architectures), x86 is handily the most widely recognized. It was first designed by Intel in 1974 in for an 8-bit processor (the Intel 8080), and over the years it was extended to 16-bit form (1978, Intel 8086), then to 32-bit form (1985, Intel 80386), and then to 64-bit form (2003, AMD Opteron). Today, processors supporting IA32 are now manufactured by Intel, AMD, and VIA, and they can be found in most personal computers.

Another well-known ISA today is the PowerPC. Apple's Macintosh computers used these processors until 2006, when Apple switched their computers to the x86 line of processors. But PowerPC remains in common use for applications such as automobiles and gaming consoles (including the Wii, Playstation 3, and XBox 360).

But the ISA that we'll study comes from a company called ARM. (Like other successful ISAs, ARM's ISA has grown over the years. We'll examine version 4T.) Processors supporting ARM's ISA are distributed quite widely, usually for low-power devices such as cellphones, digital music players, and handheld game systems. The iPhone, Kindle and Nintendo DS are all prominent examples of devices that incorporate an ARM processor.

There are several reasons for examining ARM's ISA rather than IA32.

- Assembly language programming is rarely used for more powerful computing systems, since it's far easier to program in a high-level programming language. But for small devices, assembly language programming remains important: Due to power and price constraints, the devices have very few resources, and developers can use assembly language to use these resources as efficiently as possible.
- The multiple extensions to the IA32 architecture lead it to be far too complicated for us to really understand thoroughly.
- IA32 dates from the 1970's, which was a completely different era in computing. ARM is more representative of more modern ISA designs.

2. ARM assembly basics

We'll now turn to examining ARM's ISA.

2.1. A simple program: Adding numbers

Let's start our introduction using a simple example. Imagine that we want to add the numbers from 1 to 10. We might do this in C as follows.

```
int total;
int i;

total = 0;
for (i = 10; i > 0; i--) {
    total += i;
}
```

The following translates this into the instructions supported by ARM's ISA.

```
        MOV  R0, #0           ; R0 accumulates total
        MOV  R1, #10          ; R1 counts from 10 down to 1
again   ADD  R0, R0, R1
        SUBS R1, R1, #1
        BNE  again
halt    B    halt             ; infinite loop to stop computation
```

You'll notice the mentions of `R0` and `R1` in the assembly language program. These are references to **registers**, which are places in a processor for storing data during computation. The ARM processor includes 16 easily accessible registers, numbered `R0` through `R15`. Each stores a single 32-bit number. Note that though registers store data, they are very separate from the notion of *memory*: Memory is typically much larger (kilobytes or often gigabytes), and so it typically exists outside of the processor. Because of memory's size, accessing memory takes more time than accessing registers — typically about 10 times as long. Thus, assembly language programming tends to focus on using registers when possible.

Because each line of an assembly language program corresponds directly to machine language, the lines are highly restricted in their format. You can see that each line consists of two parts: First is the **opcode** such as `MOV` that is an abbreviation indicating the type of operation; and after it comes arguments such as “`R0, #0`”. Each opcode has strict requirements on the allowed arguments. For example, a `MOV` instruction must have exactly two arguments: the first must identify a register, and the second must provide either a register or a constant (prefixed by a ‘`#`’). A constant placed directly in an instruction is called an **immediate**, since it is immediately available to the processor when reading the instruction.

In the above assembly language program, we first use the `MOV` instruction to initialize `R0` at 0 and `R1` at 10. The `ADD` instruction computes the sum of `R0` and `R1` (the second and third arguments) and places the result into `R0` (the first argument); this corresponds to the `total += i;` line of the equivalent C program. The subsequent `SUBS` instruction decreases `R1` by 1.

To understand the next instruction, we need to understand that in addition to the registers `R0` through `R15`, the ARM processor also incorporates a set of four “flags,” labeled the zero flag (Z), the negative flag (N), the carry flag (C), and the overflow flag (V). Whenever an arithmetic instruction has an `S` at its end, as **SUBS** does, these flags will be updated based on the result of the computation. In this case, if the result of decreasing `R1` by 1 results in 0, the Z flag will become 1; the N, C, and V flags are also updated, but they're not pertinent to our discussion of this code.

The following instruction, **BNE**, will check the Z flag. If the Z flag is not set (i.e., the previous subtraction gives a nonzero result), then **BNE** arranges the processor so that the next instruction executed is the **ADD** instruction, labeled `again`; this leads to repeating the loop with a smaller value of `R1`. If the Z flag is set, the processor will simply continue on to the next instruction. (**BNE** stands for Branch if Not Equal. The name comes from imagining that we want to check whether two numbers are equal. One way to do this using ARM's ISA would be to first tell the processor to subtract the two numbers; if the difference is zero, then the two numbers must be equal, and the zero flag will be 1. them results in zero, which would set the zero flag.)

The final instruction, **B**, always branches back to the named instruction. In this program, the instruction names itself, effectively halting the program by putting the computer into a tight infinite loop.

2.2. Another example: Hailstone sequence

Now, let's consider the *hailstone sequence*. Given an integer *n*, we repeatedly want to apply the following procedure.

```
$iters \leftarrow 0$
while $n \neq 1$:
   $iters \leftarrow iters + 1$
   if $n$ is odd:
      $n \leftarrow 3 \cdot n + 1$
   else:
      $n \leftarrow n / 2$
```

For example, if we start with 3, then since this is odd our next number is $3 \cdot 3 + 1 = 10$. This is even, so our next number is $10 / 2 = 5$. This is odd, so our next number is $3 \cdot 5 + 1 = 16$. This is even, so we then go to 8, which is still even, so we go to 4, then 2, and 1.

In translating this to ARM's assembly language, we must confront the fact that ARM lacks any instructions related to division. (Designers felt division too rarely necessary to merit wasting transistors on the complex circuit that it requires.) Fortunately, the division in this algorithm is relatively simple: We merely divide *n* by 2, which can be done with a right shift.

ARM has an unusual approach to shifting: We have already seen that every basic arithmetic instruction, the final argument can be a constant (as in `SUBS R1, R1, #1`) or a register (as in `ADD R0, R0, R1`). But when the final argument is a register, we can optionally add a shift distance: For instance, the instruction "`ADD R0, R0, R1, LSL #1`". says to add a left-shifted version of `R1` before adding it to `R0` (while `R1` itself remains unchanged). The ARM instruction set supports four types of shifting:

`LSL`	logical shift left
`LSR`	logical shift right
`ASR`	arithmetic shift right
`ROR`	rotate right

The shift distance can be an immediate between 1 and 32, or it can be based on a register value: "`MOV R0, R1, ASR R2`" is equivalent to "`R0 = R1 >> R2`".

In translating our pseudocode to assembly language, we'll find the shift operations useful both for multipling *n* by 3 (computed as $n + (n \ll 1)$) and for dividing *n* by 2 (computed as $n \gg 1$). We'll also need to deal with testing whether whether *n* is odd. We can do this by testing whether *n*'s 1's bit is set, which we can accomplish using the `ANDS` instruction to perform a bitwise AND with 1. The `ANDS` instruction sets the Z flag based on whether the result is 0. If the result is 0, then this means that the 1's bit of *n* is 0, and so *n* is even.

```
        MOV  R0, #5          ; R0 is current number
        MOV  R1, #0          ; R1 is count of number of iterations
again   ADD  R1, R1, #1      ; increment number of iterations
        ANDS R0, R0, #1      ; test whether R0 is odd
        BEQ  even
        ADD  R0, R0, R0, LSL #1 ; if odd, set R0 = R0 + (R0 << 1) + 1
        ADD  R0, R0, #1      ; and repeat (guaranteed R0 > 1)
        B    again
even    MOV  R0, R0, ASR #1 ; if even, set R0 = R0 >> 1
        SUBS R7, R0, #1      ; and repeat if R0 != 1
        BNE  again
halt    B    halt            ; infinite loop to stop computation
```

2.3. Another example: Adding digits

Let's look at another example. Here, suppose that we want to add the digits of a positive number; for example, given the number 1,024, we would want to compute 1 + 0 + 2 + 4, which is 7. The obvious way to express this in C is as follows.

```
    total = 0;
    while (i > 0) {
        total += i % 10;
        i /= 10;
    }
```

It's difficult to translate this into ARM's ISA, though, since the ARM lacks any instruction for dividing values. However, we can use a clever trick to perform this division using multiplication: If we take a number and multiply by 2^{32} / 10, the upper 32 bits of the product tell us the result of dividing the original number by 10. This insight leads to the following alternative way of summing the digits in a number.

```
    base = 0x1999999A;
    total = 0;
    while (i > 0) {
        iDiv10 = (i * base) >> 32;
        total += i - iDiv10 * 10;
        i = iDiv10;
    }
```

In translating this into assembly code, we have to confront two issues. The more obvious is determining which instruction to use to perform the multiplication. Here, we want to use the `UMULL` instruction (Unsigned MULtiply Long), which interprets two registers as unsigned 32-bit numbers, and places the 64-bit product of the registers' values into two different registers. The below example illustrates.

```
UMULL R4, R5, R0, R2   ; computes R0 * R2, placing lower 32 bits in R4, upper
32 in R5
```

The less obvious issue we have to confront is that of placing 0x1999999A into a register. You might be tempted at first to use `MOV`, but this instruction has a major limitation: Any immediate value must be rotated by an even number of places to reach an eight-bit value. For numbers between 0 and 255, this is not a problem; nor it is a problem for 1,024, since 0x400 can be achieved by rotating 1 left 12 places. But there's no way to do this for 0x1999999A. The solution we'll use is to load each byte separately, joining them using the `ORR` instruction, which computes the bitwise OR of two values.

```
        MOV R0, #1024           ; R0 is input, decreases by factors of 10
        MOV R1, #0              ; R1 is sum of digits
        MOV R2, #0x19000000     ; R2 is constantly 0x1999999A
        ORR R2, R2, #0x00990000
        ORR R2, R2, #0x00009900
        ORR R2, R2, #0x0000009A
        MOV R3, #10             ; R3 is constantly 10
loop    UMULL R4, R5, R0, R2    ; R5 is R0 / 10
        UMULL R4, R6, R5, R3    ; R4 is now 10 * (R0 / 10)
        SUB R4, R0, R4          ; R5 is now one's digit of R0
        ADD R1, R1, R4          ; add it into R1
        MOVS R0, R5
        BNE loop
halt    B halt
```

By the way, you may sometimes want to place a small negative number like −10 into a register. You can't use `MOV` to accomplish this, because its two's-complement representation is 0xFFFFFFF6, which can't be rotated into an 8-bit number. If it happens that to know that some register holds the number 0, then you could use `SUB`. But if it doesn't, then the `MVN` (MoVe Not) instruction is useful: It places the bitwise NOT of its argument into the destination register. So to get −10 into `R0`, we can use “`MVN R0, #0x9`”.

2.4. Summary of instructions so far

The ARM includes sixteen “basic” arithmetic instructions, numbered 0 through 15. All sixteen are listed below, with the functionality summarized by the relevant C operator. (The number at the beginning of each line is used in translating the instructions into machine language. There's no reason for programmers to memorize this correspondence, though: After all, this is why we have assemblers.)

Figure 1: ARM's basic arithmetic instructions

0. `AND` *regd, rega, argb*	*regd* ← *rega* & *argb*
1. `EOR` *regd, rega, argb*	*regd* ← *rega* ^ *argb*
2. `SUB` *regd, rega, argb*	*regd* ← *rega* − *argb*
3. `RSB` *regd, rega, argb*	*regd* ← *argb* - *rega*
4. `ADD` *regd, rega, argb*	*regd* ← *rega* + *argb*
5. `ADC` *regd, rega, argb*	*regd* ← *rega* + *argb* + *carry*
6. `SBC` *regd, rega, argb*	*regd* ← *rega* − *argb* − !*carry*
7. `RSC` *regd, rega, argb*	*regd* ← *argb* − *rega* − !*carry*
8. `TST` *rega, argb*	set flags for *rega* & *argb*
9. `TEQ` *rega, argb*	set flags for *rega* ^ *argb*
10. `CMP` *rega, argb*	set flags for *rega* − *argb*
11. `CMN` *rega, argb*	set flags for *rega* + *argb*
12. `ORR` *regd, rega, argb*	*regd* ← *rega* | *argb*
13. `MOV` *regd, arg*	*regd* ← *arg*

14. `BIC` *regd, rega, argb*	*regd* ← *rega* & ~*argb*
15. `MVN` *regd, arg*	*regd* ← ~*argb*

Except for `TST`, `TEQ`, `CMP`, and `CMN`, all instructions may have an `S` postfixed to the opcode to signify that the operation should set the flags. For `TST`, `TEQ`, `CMP`, and `CMN`, the `S` is implicit: The instructions don't change any general-purpose registers, so the only point in performing the instruction is to set the flags.

We've also seen three other opcodes that aren't in the above of basic arithmetic instructions: `UMULL` is a “non-basic” arithmetic instruction, and `B` and `BNE` aren't arithmetic instructions.

2.5. Condition codes

Each ARM instruction may incorporate a **condition code** specifying that the operation should take place only when certain combinations of the flags hold. You can specify the condition code by including it as part of the opcode. It usually comes at the end of the opcode, but it precedes the optional `S` on the basic arithmetic instructions. The name for the condition codes is based on the supposition that the flags were set based on a `CMP` or `SUBS` instruction.

Figure 2: ARM's condition codes

0. `EQ`	equal	`Z`
1. `NE`	not equal	`!Z`
2. `CS` or `HS`	carry set / unsigned higher or same	`C`
3. `CC` or `LO`	carry clear / unsigned lower	`!C`
4. `MI`	minus / negative	`N`
5. `PL`	plus / positive or zero	`!N`
6. `VS`	overflow set	`V`
7. `VC`	overflow clear	`!V`
8. `HI`	unsigned higher	`C && !Z`
9. `LS`	unsigned lower or same	`!C || Z`
10. `GE`	signed greater than or equal	`N == V`
11. `LT`	signed less than	`N != V`
12. `GT`	signed greater than	`!Z && (N == V)`
13. `LE`	signed greater than or equal	`Z || (N != V)`
14. `AL` or omitted	always	`true`

The only instance of this condition code we have seen so far is the `BNE` instruction: In this case, we have a `B` instruction for branching, but the branch only takes place if the Z flag is 0.

But ARM's ISA allows us to apply condition codes to other opcodes, too. For example, `ADDEQ` says to perform an addition if the Z flag is 1. One common scenario using condition codes on non-branch instructions is in computing the greatest common divisor of two numbers using Euclid's GCD algorithm.

```
    a = 40;
    b = 25;
    while (a != b) {
        if (a > b) a -= b;
        else       b -= a;
    }
```

The traditional translation to assembly language would use condition codes only on branch instructions.

```
        MOV R0, #40       ; R0 is a
        MOV R1, #25       ; R1 is b
again   CMP R0, R1
        BEQ halt
        BLT isLess
        SUB R0, R0, R1
        B again
isLess  SUB R1, R1, R0
        B again
halt    B halt
```

However, the following is a much shorter and more efficient translation.

```
        MOV R0, #40       ; R0 is a
        MOV R1, #25       ; R1 is b
again   CMP R0, R1
        SUBGT R0, R0, R1
        SUBLT R1, R1, R0
        BNE again
halt    B halt
```

This is more efficient for two reasons. More obviously, the number of instructions executed per iteration is smaller (four versus five). But the other reason comes from the fact that modern processors “pre-fetch” the following instruction while executing the current instruction. However, branches disrupt this process since the location of the next instruction can't be known certainly. The second translation involves many fewer branch instructions, so it will have fewer problems with pre-fetching instructions.

3. Memory

We've seen how to build assembly programs that perform basic numerical computation. We'll now turn to examining how assembly programs can access memory.

3.1. Basic memory instructions

The ARM supports memory access via two instructions, **LDR** and **STR**. The **LDR** instruction loads data out of memory, and **STR** stores data into memory. Each takes two arguments. The first argument is the data register: For an **LDR** instruction, the loaded data is placed into this register; for an **STR** instruction, the data found in this register is stored into memory. The second argument indicates the register that contains the memory address being accessed; it will be written using the register name enclosed in brackets. (In Section 3.2, we will see that there are other options for how this second argument can be written.)

For an example of how these instructions work, let's suppose we want a assembly program fragment that adds the integers in an array. We imagine that `R0` holds the address of the first integer of the array, and `R1` holds the number of integers in the array.

```
addInts MOV R4, #0
addLoop LDR R2, [R0]
        ADD R4, R4, R2
        ADD R0, R0, #4
        SUBS R1, R1, #1
        BNE addLoop
```

In this fragment, we use `R4` to hold the sum of the integers so far. In the **LDR** instruction, we look into `R0` for a memory address and load the data found at that address into `R2`. We then add this value into `R4`. Then, we move `R0` so that it contains the memory address of the next integer in the array; we increase `R0` by four because each integer consumes four bytes of memory. Finally, we decrement `R1`, which is the number of integers left to read from the array, and we repeat the process if there are integers remaining.

Both **LDR** and **STR** load and store 32-bit values. There are also instructions for working with 8-bit values, **LDRB** and **STRB**; these are useful primarily for working with strings. Below is an implementation of C's `strcpy` function; we imagine that `R0` holds the address of the first character of the destination array, and that R1 holds the address of the first character of the source string. We want to keep copying until we copy the terminating NUL character (ASCII 0).

```
strcpy  LDRB R2, [R1]
        STRB R2, [R0]
        ADD R0, R0, #1
        ADD R1, R1, #1
        TST R2, R2      ; repeat if R2 is nonzero
        BNE strcpy
```

3.2. Addressing modes

In the previous section's examples, we provided the address by enclosing a register's name in brackets. But the ARM allows several other ways of indicating the memory address, too. Each such technique is called an **addressing mode**; the technique of simply naming a register holding a memory address is one such addressing mode, called *register* addressing, but there are others.

One of these others is *scaled register offset*, where we include in the brackets a register, another register, and a shift value. To compute the memory address to access, the processor takes the first register, and adds to it the second register shifted according to the shift value. (Neither of the registers mentioned in brackets change values.) This addressing mode is useful when accessing an array where you know the array index. We can modify our earlier routine for adding the integers in an array to take advantage of this addressing mode.

```
addInts MOV R4, #0
addLoop SUBS R1, R1, #1
        LDR R2, [R0, R1, LSL #2]
        ADD R4, R4, R2
        BNE addLoop
```

With each iteration of the loop, we first decrement our loop index R1. Then we retrieve the element at that entry of the array using a scaled register offset: We use R0 as our base, and we add to it R1 shifted left two places. We shift R1 left two places so that R1 is multiplied by four; after all, each integer in the array is four bytes long. After adding the loaded value into R4, which accumulates the total, we repeat the loop if R1 hasn't reached 0 yet.

Beyond using a different addressing mode, this version of the code is slightly different from our original implementation in three ways. First, it loads the numbers in the array in reverse order — that is, it loads the last number in the array first. Second, R0 remains unaltered in the course of the fragment. And finally, it will be somewhat faster since it has one less instruction per loop iteration.

Immediate post-indexed addressing is another addressing mode. To indicate this mode in assembly language, we follow the brackets with a comma and a positive or negative immediate. In executing the instruction, the processor still accesses the memory address found in the register, but after accessing the memory the address register is increased or decreased according to the immediate.

Our `strcpy` implementation is a useful example where immediate post-indexed addressing is useful: After we store to `R0`, we want `R0` to increase by 1 for the following iteration; and similarly, after we load from `R1`, we want `R1` to increase by 1. We can use immediate post-indexed addressing to avoid the two `ADD` instructions of our earlier version.

```
strcpy  LDRB R2, [R1], #1
        STRB R2, [R0], #1
        TST R2, R2       ; repeat if R2 is nonzero
        BNE strcpy
```

In total, the ARM processor supports ten addressing modes.

`[Rn, #±imm]`	*Immediate offset* Address accessed is *imm* more/less than the address found in R*n*. R*n* does not change.
`[Rn]`	*Register* Address accessed is value found in R*n*. This is just shorthand for `[Rn, #0]`.
`[Rn, ±Rm, shift]`	*Scaled register offset* Address accessed is sum/difference of the value in R*n* and the value in R*m* shifted as specified. R*n* and R*m* do not change values.
`[Rn, ±Rm]`	*Register offset* Address accessed is sum/difference of the value in R*n* and the value in R*m*. R*n* and R*m* do not change values. This is just shorthand for `[Rn, ±Rm, LSL #0]`.
`[Rn, #±imm]!`	*Immediate pre-indexed* Address accessed is as with *immediate offset* mode, but R*n*'s value updates to become the address accessed.

`[Rn, ±Rm, shift]!`	*Scaled register pre-indexed* Address accessed is as with *scaled register offset* mode, but R*n*'s value updates to become the address accessed.
`[Rn, ±Rm]!`	*Register pre-indexed* Address accessed is as with *register offset* mode, but R*n*'s value updates to become the address accessed.
`[Rn], #±imm`	*Immediate post-indexed* Address accessed is value found in R*n*, and then R*n*'s value is increased/decreased by *imm*.
`[Rn], ±Rm, shift`	*Scaled register post-indexed* Address accessed is value found in R*n*, and then R*n*'s value is increased/decreased by R*m* shifted according to *shift*.
`[Rn], ±Rm`	*Register post-indexed* Address accessed is value found in R*n*, and then R*n*'s value is increased/decreased by R*m*. This is just shorthand for `[Rn], ±Rm, LSL #0`.

For those addressing modes involving a shift, the shift technique is as with the arithmetic instructions (`LSL`, `LSR`, `ASR`, `ROR`, `RRX`). But the shift distance cannot be according to a register: The distance must be an immediate.

3.3. Initializing memory

We often want to reserve memory for holding data in a program. To do this, we use **directives**: directions for the assembler to do something other than simply translate an assembly language instruction into its corresponding machine code. One useful directive is `DCD`, which inserts one or more 32-bit numerical values into the machine code output. (`DCD` cryptically stands for Define Constant Double-words.)

```
primes  DCD   2, 3, 5, 7, 11, 13, 17, 19
```

In this example, we've created the label `primes`, which will correspond to the address where 2 is placed into memory. In the following four bytes is placed the integer 3, then 5, and so on.

In our program, we would want to load the address of the array into a register; to do this, we add `primes` into the program counter `PC` (which is synonymous with `R15`). The below fragment loads the fifth prime (11) into `R1`.

```
ADD R0, PC, #primes  ; load address of primes[0] into R0
        LDR R1, [R0, #16]     ; load primes[4] into R1
```

Another directive worth mentioning is **`DCB`**, for loading bytes into memory. Thus, we could write the following.

```
primes  DCB   2, 3, 5, 7, 11, 13, 17, 19
```

However, we are using just one byte for each number, so we can only include numbers between −128 and 127. We can also include a string in the list; each character of the string will occupy one byte of memory.

```
greet   DCB   "hello world\n", 0
```

Notice how we included 0 after the string. Without this, the string won't be terminated by the NUL character.

One more directive worth noting here is the percent sign **`%`**. This is useful when you wish you reserve a block of memory, but you don't care about the memory's initial value.

```
array   % 120  ; reserve 120 bytes of memory, which can hold 30 ints
```

3.4. Multiple-register memory instructions

The ARM ISA also includes instructions allowing several values to be loaded or stored in the same instruction. The **`LDMIA`** instruction is one such instruction: It allows loading into multiple registers starting at an address named in another register. In the below example of its usage, we take our code for adding the integers of an array, and we modify it using **`LDMIA`** so that it processes four integers with each iteration of the loop. This strategy allows the program to run using fewer instructions, at the expense of more complexity.

```
; R0 holds address of first integer in array
; R1 holds array's length; fragment works only if length is multiple of 4
addInts MOV R4, #0
addLoop LDMIA R0!, { R5-R8 }
        ADD R5, R5, R6
        ADD R7, R7, R8
        ADD R4, R4, R5
        ADD R4, R4, R7
        SUBS R1, R1, #4
        BNE addLoop
```

In executing the **`LDMIA`** instruction above, the ARM processor looks into the `R0` register for an address. It loads into `R5` the four bytes starting at that address, into `R6` the next four bytes, into `R7` the next four bytes, and into `R8` the next four bytes. Meanwhile, `R0` is stepped forward by 16 bytes, so with the next iteration the **`LDMIA`** instruction will load the next four words into the registers.

Inside the braces can be any list of registers, using dashes to indicate ranges of registers, and using commas to separate ranges. Thus, the instruction `LDMIA R0!, { R1-R4, R8, R11-R12 }` will load seven words from memory. The order in which the registers are listed is not significant; even if we write `LDMIA R0!, { R11-R12, R8, R1-R4 }`, `R1` will receive the first word loaded from memory.

The exclamation point following `R0` in our example may be omitted; if omitted, then the address register is not altered by the instruction. That is, `R0` would continue pointing to the first integer in the array. In our example above, we want `R0` to change so that it is pointing to the next block of four integers for the next iteration, so we included the exclamation point.

Another instruction is **`STMIA`**, which stores several registers into memory. In the following example, we shift every number in an array into the next spot; thus, the array <2,3,5,7> becomes <0,2,3,5>.

```
; R0 holds address of first integer in array
; R1 holds array's length; fragment works only if length is multiple of 4
shift   MOV R4, #0
shLoop  LDMIA R0, { R5-R8 }
        STMIA R0!, { R4-R7 }
        MOV R4, R8
        SUBS R1, R1, #4
        BNE shLoop
```

Notice how the **`LDMIA`** instruction omits the exclamation point so that `R0` isn't modified. This is so that **`STMIA`** stores into the same range of addresses that were just loaded into the registers. The **`STMIA`** instruction has the exclamation point because `R0` must be modified in preparation for the next iteration of the loop.

The ARM processor includes four variants of the multiple-load and multiple-store instructions; the `LDM` and `STM` abbreviations must always indicate one of these four variants.

`LDMIA`, **`STMIA`** *Increment after*

We start loading from the named address and into increasing addresses.

`LDMIB`, **`STMIB`** *Increment before*

We start loading from four more than the named address and into increasing addresses.

`LDMDA`, **`STMDA`** *Decrement after*

We start loading from the named address and into decreasing addresses.

`LDMDB`, **`STMDB`** *Decrement before*

We start loading from four less than the named address and into decreasing addresses.

Across all four modes, the highest-numbered register always corresponds to the highest address in memory. Thus, the instruction `LDMDA R0, { R1-R4 }` will place `R4` into the address named by `R0`, `R3` into `R0` – 4, and so on.

As we'll see in studying subroutines, the different variants are particularly useful when we want to use a block of unused memory as a stack.