4

ARM® Assembly Language

In this chapter, we will study the ARM instruction set. As of 2012, this instruction set is the most widely used instruction set in smart phones, and tablets. It has more than 90% market share[1] in this space. ARM processors are also one of the most popular processors in hard disk drives, and set top boxes for televisions. Hence, for any student of computer architecture it is very important to learn about the ARM instruction set because it will prove to be useful in programming the mobile and handheld devices of the future.

The ARM instruction set is a 32-bit instruction set. This means that the sizes of all registers are 32 bits, and the size of the memory address is equal to 32 bits. It is a RISC instruction set with a very regular structure. Each instruction is encoded into a string of exactly 32 bits like *SimpleRisc* . All arithmetic and logical operations, use only register operands, and lastly all the communication between registers and memory happens through two data transfer instructions – load and store.

4.1 The ARM® Machine Model

ARM assembly language assumes a machine model similar to that explained in Section 3.2.1 for *SimpleRisc* . For the register file, it assumes that there are 16 registers that are visible to the programmer at any point of time. All the registers in ARM are 32 bits or 4 bytes wide.

The registers are numbered from $r0$ to $r15$. Registers $r11 \dots r15$ are known by certain mnemonics also as shown in Table 4.1. $r11$ is the frame-pointer. It points to the top of the activation block. $r12$ is a scratch register that is not meant to be saved by the caller or the callee. $r13$ is the stack pointer. It is important to understand that $r11$ and $r12$ are assigned a special connotation by the GNU compiler collection. They are not assigned special roles by the ARM ISA.

[1] Most of the ARM code running on processors is actually written in the Thumb-2 ARM ISA. The Thumb-2 ISA is essentially a recoding (or a simpler variant) of the ISA presented in this chapter. Hence, it is necessary for readers to get a thorough understanding of the material that follows.

Register	Abbrv.	Name
$r11$	fp	frame pointer
$r12$	ip	intra-procedure-call scratch register
$r13$	sp	stack pointer
$r14$	lr	link register
$r15$	pc	program counter

Table 4.1: Registers with special names in ARM

Let us differentiate between generic registers and registers with special roles. Registers $r0 \ldots r12$ are generic. The programmer and the compiler can use them in any way they like. However, the registers $r13$(sp), $r14$(lr) and $r15$(pc) have special roles. *sp* is the stack pointer, *lr* is the return address register, and *pc* is the program counter. In this chapter, we shall use the little endian version of the ARM ISA, and we shall describe the syntax of the assembly language used by the GNU ARM Assembler [arm, 2000].

4.2 Basic Assembly Instructions

4.2.1 Simple Data Processing Instructions

Register Transfer Instructions

The simplest type of assembly instructions transfer the value of one register into another, or store a constant in a register. There are two instructions in this class – *mov* and *mvn*. Their semantics are shown in Table 4.2. Note that we always prefix an immediate with '#' in ARM assembly.

Semantics	Example	Explanation
mov *reg*, (*reg*/*imm*)	mov r1, r2	r1 ← r2
	mov r1, #3	r1 ← 3
mvn *reg*, (*reg*/*imm*)	mvn r1, r2	r1 ← ∼ r2
	mvn r1, #3	r1 ← ∼ 3

Table 4.2: Semantics of the move instructions

The register based *mov* instruction simply moves the contents of $r2$ to register $r1$. Alternatively, it can store an immediate in a register. In Table 4.2, the *mvn* instruction flips every bit in the 32-bit register $r2$, and then transfers the contents of the result to $r1$. The $\sim$ symbol represents logical complement. For example, the complement of the 4-bit binary value, 0110, is 1001. The *mov* and *mvn* instructions take two inputs. These instructions are examples of 2-address format instructions in ARM.

Arithmetic Instructions

The simplest instructions in this class are *add*, *sub*, *rsb* (reverse subtract). Their semantics are given in Table 4.3. The second operand can also be an immediate.

Semantics	Example	Explanation
add *reg*, *reg*, (*reg*/*imm*)	add r1, r2, r3	r1 ← r2 + r3
sub *reg*, *reg*, (*reg*/*imm*)	sub r1, r2, r3	r1 ← r2 - r3
rsb *reg*, *reg*, (*reg*/*imm*)	rsb r1, r2, r3	r1 ← r3 - r2

Table 4.3: Semantics of add and subtract instructions

Example 37
Write an ARM assembly program to compute: 4+5 - 19. Save the result in r1.

Answer: *Simple yet suboptimal solution.*

```
mov r1, #4
mov r2, #5
add r3, r1, r2
mov r4, #19
sub r1, r3, r4
```

Optimal solution.

```
mov r1, #4
add r1, r1, #5
sub r1, r1, #19
```

Logical Instructions

Semantics	Example	Explanation
and *reg*, *reg*, (*reg*/*imm*)	and r1, r2, r3	r1 ← r2 AND r3
eor *reg*, *reg*, (*reg*/*imm*)	eor r1, r2, r3	r1 ← r2 XOR r3
orr *reg*, *reg*, (*reg*/*imm*)	orr r1, r2, r3	r1 ← r2 OR r3
bic *reg*, *reg*, (*reg*/*imm*)	bic r1, r2, r3	r1 ← r2 AND (∼ r3)

Table 4.4: Semantics of logical instructions

ARM's bitwise logical instructions are shown in Table 4.4. *and* computes a bit-wise AND, *eor* computes an exclusive OR, *orr* computes a regular bit-wise OR, and the *bic*(bit-clear) instruction clears off the bits in $r2$ that are specified in $r3$. Like arithmetic instructions, the second operand can be an immediate.

Example 38
Write an ARM assembly program to compute: $\overline{A \vee B}$, *where* A *and* B *are 1 bit Boolean values. Assume that* $A = 0$ *and* $B = 1$. *Save the result in* $r0$.

Answer:

```
mov r0, #0x0
orr r0, r0, #0x1
mvn r0, r0
```

Multiplication Instructions

We shall introduce four multiply instructions with varying degrees of complexity. The fundamental issue with multiplication is that if we are multiplying two 32-bit numbers, then the result will require 64 bits. The reason is that the largest unsigned 32-bit number is $2^{32} - 1$. Consequently, when we try to square this number, our result is approximately 2^{64}. We would thus need a maximum of 64 bits.

ARM has two 32-bit multiplication instructions that truncate the result to 32 bits – *mul* and *mla*. They ignore the rest of the bits. *mul* multiplies the values in two registers and stores the result in a third register. *mla* (multiply and accumulate) is in the 4-address format. It multiplies the values of two registers, and adds the result to the value stored in a third register (see Table 4.5). The advantage of the *mla* instruction is that it makes it possible to represent code sequences of the form ($d = a + b * c$) with one instruction. Such instructions are extremely useful when it comes to implementing linear algebra kernels such as matrix multiplication.

Semantics	Example	Explanation
mul *reg*, *reg*, *reg*	mul r1, r2, r3	r1 $\leftarrow$ r2 $\times$ r3
mla *reg*, *reg*, *reg*, *reg*	mla r1, r2, r3, r4	r1 $\leftarrow$ r2 $\times$ r3 + r4
smull *reg*, *reg*, *reg*, *reg*	smull r0, r1, r2, r3	$\underbrace{r1\ r0}_{64} \leftarrow$ r2 $\times_{signed}$ r3
umull *reg*, *reg*, *reg*, *reg*	umull r0, r1, r2, r3	$\underbrace{r1\ r0}_{64} \leftarrow$ r2 $\times_{unsigned}$ r3

Table 4.5: Semantics of multiply instructions

In this chapter, we shall introduce two instructions that store the entire 64-bit result in two registers. The *smull* and *umull* instructions perform signed and unsigned multiplication

respectively on two 32-bit values to produce a 64-bit result. Their semantics is shown in Table 4.5. $r0$ contains the lower 32 bits, and $r1$ contains the upper 32 bits.

For all the multiply instructions that we have introduced, all the operands need to be registers. Secondly, the first source register, should not be the same as the destination register.

Example 39
Compute $12^3 + 1$, *and save the result in r3.*
Answer:

```
/* load test values */
mov r0, #12
mov r1, #1

/* perform the logical computation */
mul r4, r0, r0       @ 12*12
mla r3, r4, r0, r1   @ 12*12*12 + 1
```

Division Instructions

Newer versions of the ARM ISA have introduced two integer division instructions, *sdiv* and *udiv*. The former is used for signed division and the latter is used for unsigned division (see Table 4.6). Both of them compute the quotient. The remainder can be computed by subtracting the product of the dividend and the quotient from the dividend.

Semantics	Example	Explanation
sdiv *reg*, *reg*, *reg*	sdiv r1, r2, r3	r1 ← r2 ÷ r3 (signed)
udiv *reg*, *reg*, *reg*	udiv r1, r2, r3	r1 ← r2 ÷ r3 (unsigned)

Table 4.6: Semantics of divide instructions

4.2.2 Advanced Data-Processing Instructions

Let us consider the generic format of 3-address data-processing instructions.

```
instruction <destination register> <register operand 1> <operand 2>
```

Likewise, the generic format for 2 address data processing instructions is

```
instruction <register operand 1> <operand 2>
```

Up till now, we have been slightly quiet about $<$ *operand* 2 $>$. It can be a register operand, an immediate, or a special class of operands called – *shifter operands*. The first two classes are

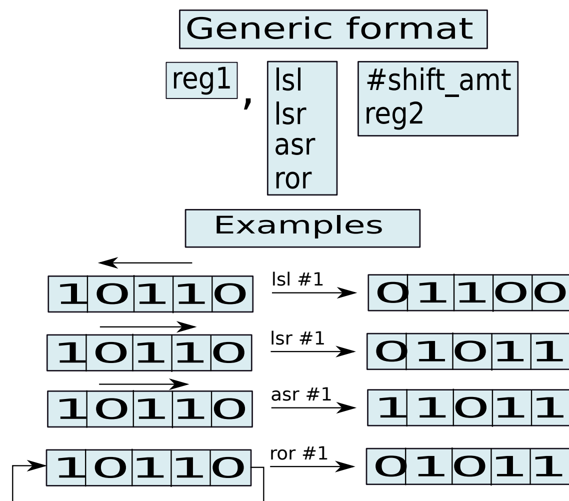

Figure 4.1: Format of shifter operands

intuitive. Let us describe shifter operands in this section. Their generic format is shown in Figure 4.1.

A shifter operand contains two parts. This first part is a register, and the latter part specifies an operation to be performed on the value in the register. The ARM instruction set defines four such operations – *lsl* (logical shift left), *lsr* (logical shift right), *asr* (arithmetic shift right), and *ror* (rotate right). These operations are collectively called shift and rotate instructions.

Shift and Rotate Instructions

A logical left shift operation is shown in Figure 4.1. In this example, we are shifting the value 10110 one place to the left. We need to shift in an extra 0 at the LSB position. The final result is equal to 01100. A left shift operation is present in most programming languages including C and Java. It is denoted by the following symbol: $\ll$. Note that shifting a word (4 byte number) by k positions to the left is equivalent to multiplying it by 2^k. This is in fact a quick way of multiplying a number by a power of 2.

Let us now consider the right shift operation. Unlike the left shift operation, this operation comes in two variants. Let us first consider the case of unsigned numbers. Here, we treat a word as a sequence of 32 bits. In this case, if we shift the bits 1 position to the right, we fill the MSB with a 0. This operation is known as – logical shift right (see Figure 4.1). Note that shifting a number right by k places is usually the same as dividing it by 2^k. The right shift operation in C or Java is $\gg$.

If we consider a signed number, then we need to use the arithmetic right shift (*asr*) operation. This operation preserves the sign bit. If we shift a number right using *asr* by one position, then we fill the MSB with the previous value of the MSB. This ensures that if we shift a negative number to the right, the number still remains negative. In a four bit number system, if we shift 1010 to the right by 1 place using *asr*, then we get 1101. The original number is -6, and the shifted number is equal to -3. We thus see that arithmetic right shift divides a signed

number by a power of two. Note that using the right shift operations for odd numbers is tricky. Let us consider the representation of -5 in a 4-bit number system. It is 1011. After performing an arithmetic right shift, the result is equal to 1101, which is equal to -3 in decimal. Whether we consider -5/2 = -3 as a correct answer or not depends on the semantics of the programming language.

The right rotate operation performs a right shift on the number. However, it fills the MSB with the number shifted out from the rightmost end. In Figure 4.1, if we right rotate 10110, we get 01011. In this case we have moved the previous LSB (0) to the new MSB. Note that ror (right rotate) by 32 positions gives us the original value. ARM provides a special connotation for *ror #0*. It performs a right shift. It moves the value of the carry flag to the MSB, and then sets the shifted out LSB to the carry flag. This is also referred to as the *rrx* operation. This operation does not take any arguments.

Using Shifter Operands

A shifter operand of the form – r1, lsl #2 – means that we shift the value in r1 by 2 places to the left. Note that the value in r1 is not affected in this process. Likewise, an operand of the form – r1, lsr r3 – means that we shift the value in r1 to the right by the value specified in r3. We can now use the shifter operand as a valid second operand. See examples 40, and 41.

Example 40
Write ARM assembly code to compute: r1 = r2 / 4. Assume that the number stored in r1 is divisible by 4.

Answer:

```
mov r1, r2, asr #2
```

Example 41
Write ARM assembly code to compute: r1 = r2 + r3 × 4.

Answer:

```
add r1, r2, r3, lsl #2
```

Addressing Modes

We have now seen different formats of operands. An operand can either be a register, an immediate, or a shifted register.

We have up till now seen three addressing modes:

1. register addressing mode: Example, $r1$, $r2$, $r3$

2. immediate addressing mode: Example, #1, #2
3. scaled-register addressing mode: Example, $(r1, lsl\ \#2)$, $(r1, lsl\ r2)$

4.2.3 Compare Instructions

ARM has four compare instructions – *cmp*, *cmn*, *tst*, and *teq* – in the 2-address format. These instructions compare the values in the two registers and save some properties of the result of the comparison in a dedicated internal register called the *CPSR* register. Other instructions base their behavior based on the values saved in the CPSR register. This is similar to the $flags$ register in *SimpleRisc* .

The $CPSR$ register

The CPSR (Current Program Status Register) maintains some state regarding the execution of the program. It is a 32-bit register like the other registers, and is usually used implicitly. In this book, we are concerned with four bits that it stores in the positions [29-32]. They are N(Negative), Z(Zero), C(Carry), and V(Overflow). These four bits are known as *condition code flags*, or simply *flags*. It is similar to the $flags$ register in *SimpleRisc* .

There are two sets of instructions that can set CPSR flags. The first set comprises of compare instructions, and the second set includes flag setting variants of generic instructions. In either case, the rules for setting the flags are as follows:

N (Negative) This flag is set if the result is a 2's complement based signed integer. It is set to 1 if the result is negative, and 0 if it is non-negative.

Z (Zero) This flag is set to 1 if the result is zero. In a comparison operation, if the operands are equal, then this flag is also set to 1.

C (Carry)
- For an addition, the C bit is set to 1 if the result produced a carry. This can happen when there was an overflow while adding the unsigned numbers. For example, if we add -1(1111_2) and -2(1110_2), then the result is -3(1101_2), and there is a carry out at the MSB. Note that there is no real overflow, because -3 can be represented in the number system. However, if the numbers are treated as unsigned numbers, then there is an *unsigned overflow*. Consequently, we can also say that the carry bit is set if there is an unsigned overflow.
- For a subtraction, the carry bit is set to 0 if there is an unsigned underflow. For example, if we try to compute $0 - 1$, then there is no real overflow/underflow. However, $0000_2 - 0001_2$ will lead to an unsigned underflow. This basically means that when we subtract these two numbers, we will need to borrow a bit. In this case, we set the C flag to 0. Otherwise, we set it to 1.
- For logical shift operations, C is equal to the last bit shifted out of the result value.

V (Overflow) V is set to 1 when an actual signed overflow/underflow occurs. Note that in the rest of the book, we might casually refer to both overflow and underflow as just *overflow*.

Compare Instructions

ARM has four compare instructions – *cmp*, *cmn*, *tst* and *teq*. All four of them update the CPSR flags. Let us consider the *cmp* instruction. It is a 2-address instruction that takes two inputs. It essentially subtracts their values and sets the appropriate flags. For example, if the values are equal, then the zero flag is set. Later instructions can take some decisions based on these flags. For example, they might decide if they need to branch, or perform a certain computation based on the value of the zero flag. We show the semantics of all four compare instructions in Table 4.7.

Semantics	Example	Explanation
cmp *reg*, (*reg*/*imm*)	cmp r1, r2	Set flags after computing (r1 - r2)
cmn *reg*, (*reg*/*imm*)	cmn r1, r2	Set flags after computing (r1 + r2)
tst *reg*, (*reg*/*imm*)	tst r1, r2	Set flags after computing (r1 AND r2)
teq *reg*, (*reg*/*imm*)	teq r1, r2	Set flags after computing (r1 XOR r2)

Table 4.7: Semantics of compare instructions

cmn computes the flags after adding the register values, *tst* computes a bitwise AND of the two operands and then sets the flags, and *teq* tests for equality by computing an XOR (exclusive or) of the operands. For this set of instructions, the second operand can be an immediate also. Note that the compare instructions, are not the only instructions that can set the flags. Let us discuss a generic class of instructions that can set the CPSR flags.

4.2.4 Instructions that Set CPSR Flags – The 'S' Suffix

Normal instructions such as *add* and *sub* do not set the CPSR flags. However, it is possible to make any data processing instruction set the flags by adding the suffix - 's' - to it. For example, the *adds* and *subs* instructions do the regular jobs of addition and subtraction respectively, and additionally also set the CPSR flags. The rules for setting the flags are given in Section 4.2.3. Let us now see how we can use these flags.

4.2.5 Data Processing Instructions that use CPSR Flags

There are three simple data processing instructions that use the CPSR flags in their computation. They are *sbc*, *rsc*, and *adc*.

Let us now motivate this section with an example. Our basic ARM instruction format does not support 64-bit registers. Consequently, if we desire to implement the *long* data type that uses 64 bits, we need to use two registers. Let us assume that one long value is present in registers, $r2$, and $r1$. Here, $r2$ contains the upper 32 bits, and $r1$ contains the lower 32 bits. Let the second long value be present in registers $r4$, and $r3$. Let us now try to add these two long values to produce a 64-bit result, and save it in registers, $r6$ and $r5$. See Example 42.

Example 42
Add two long values stored in r2,r1 and r4,r3.

Answer:

```
adds r5, r1, r3
adc  r6, r2, r4
```

The (adds) instruction adds the values in r1 and r3. adc(add with carry) adds r2, r4, and the value of the carry flag. This is exactly the same as normal addition.

Example 43 shows how to subtract the values.

Example 43
Subtract two long values stored in r2,r1 and r4,r3.

Answer:

```
subs r5, r1, r3
sbc  r6, r2, r4
```

subs subtracts the value of r3 from the value in r1. sbc(subtract with carry) subtracts the value in r4 from the value in r2. Additionally, if the previous instruction resulted in a borrow (carry equal to 0), then it also subtracts the carry bit. This is the same as normal subtraction.

We list the semantics of the instructions in Table 4.8. Note that in the case of a subtraction the carry flag is set to 0, when there is a borrow. The *NOT* operation flips a 0 to 1, and vice versa. Lastly, *rsc* stands for – *reverse subtract with carry.*

Semantics	Example	Explanation
adc *reg*, *reg*, *reg*	adc r1, r2, r3	r1 = r2 + r3 + Carry_Flag
sbc *reg*, *reg*, *reg*	sbc r1, r2, r3	r1 = r2 - r3 - NOT(Carry_Flag)
rsc *reg*, *reg*, *reg*	rsc r1, r2, r3	r1 = r3 - r2 - NOT(Carry_Flag)

Table 4.8: Semantics of *adc*, *sbc*, and *rsc* instructions

4.2.6 Simple Branch Instructions

An ISA with just data processing instructions is very weak. We need branch instructions such that we can implement if-statements and for-loops. ARM programs primarily use three branch

instructions to do most of their work. They are: *b*, *beq*, *bne*. Their semantics are given in Table 4.9.

Semantics	Example	Explanation
b *label*	b .foo	Jump unconditionally to label .foo
beq *label*	beq .foo	Branch to .foo if the last flag setting instruction has resulted in an equality and (Z flag is 1)
bne *label*	bne .foo	Branch to .foo if the last flag setting instruction has resulted in an inequality and (Z flag is 0)

Table 4.9: Semantics of simple branch instructions

Example 44

Write an ARM assembly program to compute the factorial of a positive number (> 1) stored in r0. Save the result in r1.

Answer:

C
```c
int val = get_input();
int idx;
int prod = 1;
for (idx = 1; idx <= val ;
      idx++) {
    prod = prod * idx;
}
```

ARM assembly
```
    mov r1, #1        /* prod = 1 */
    mov r3, #1        /* idx = 1 */
.loop:
    mul r1, r3, r1    /* prod = prod * idx */
    cmp r3, r0        /* compare idx, with the input (num) */
    add r3, r3, #1    /* idx ++ */
    bne .loop         /* loop condition */
```

Let us now see, how we can use the power of branches to write some powerful programs. Let us consider the factorial function. In Example 44, we show a small program to compute the

factorial of a natural number. $r3$ is a counter that is initialised to 0. We keep on incrementing it till it matches $r0$. $r1$ represents the product. We iteratively multiply the value of $r3$ with $r1$. At the end of the set of iterations, $r1$ contains the factorial of the value given in $r0$.

Example 45
Write an assembly program to find out if a natural number stored in $r0$ is a perfect square. Save the Boolean result in $r1$.
Answer:

```
1  mov r1, #0 /* result initialised to false */
2  mov r2, #1 /* counter */
3  .loop:
4          mul r3, r2, r2
5          cmp r3, r0
6          beq .square
7          add r2, r2, #1
8          cmp r2, r0
9          bne .loop
10
11         b .exit    /* number is not a square */
12 .square:
13         mov r1, #1 /* number is a square */
14 .exit:
```

Let us show the example of another program to test if a number is a perfect square (see Example 45). $r1$ contains the result of the operation. If the number is a perfect square we set $r1$ to 1, else we set $r1$ to 0. The main loop is between lines 3 and 9. Here, we increment the value of $r2$ iteratively, and test if its square equals $r0$. If it does, we jump to .square, set $r1$ to 1, and jump to .exit. Here, we print the value (code not shown), and exit the program. We assume a hypothetical label – .exit – that is present at the end of the program (also shown in the code). The exit condition of the loop is Line 9, where we consider the result of the comparison of $r2$ and $r0$. If $r2$ is equal to $r0$, then $r0$ cannot contain a perfect square because $r0$ is at least equal to 2 at the end of any iteration.

4.2.7 Branch and Link Instruction

We can use the simple branch instructions to implement *for* loops and *if* statements. However, we need a stronger variant of the branch instruction to implement function calls. Function calls are different than regular branches because we need to remember the point in the program that the function needs to return to. ARM provides the *bl* (branch-and-link) instruction for this purpose. The semantics of this instruction is shown in Table 4.10.

Semantics	Example	Explanation
bl *label*	bl .foo	(1) Jump unconditionally to the function at .foo (2) Save the next PC (PC + 4) in the *lr* register

Table 4.10: Semantics of the branch and link instruction

The *bl* instruction jumps to the function that begins at the specified label. Note that in the ARM ISA, there is no special way for designating the start of a function. Any instruction can in principle be the start of a function. In ARM assembly, the starting instruction of a function needs to have a label assigned to it. Along with branching to the given label, the *bl* instruction also saves the value of the return address, which is equal to the current PC plus 4, into the *lr* register ($r14$). We need to add 4 over here because the size of an instruction in ARM is exactly equal to 4 bytes.

Once a function starts executing, it is expected that it will preserve the value of the return address saved in the *lr* register unless it invokes other functions. If a function invokes other functions, it needs to spill and restore registers as mentioned in Section 3.3.10. When we wish to return from a function, we need to move the value in the *lr* register to the *pc* register ($r15$). The PC will point to the instruction at the return address and execution will proceed from that point.

Example 46

Example of an assembly program with a function call.

C

```
int foo() {
    return 2;
}
void main() {
        int x = 3;
        int y = x + foo();
}
```

ARM assembly

```
foo:
    mov r0, #2
    mov pc, lr

main:
    mov r1, #3      /* x = 3 */
    bl foo          /* invoke foo */
                    /*  y = x + foo() */
    add r2, r0, r1
```

Let us take a look at Example 46. In this example, we consider a simple piece of C code that calls a function *foo* that returns a constant value of 2. It adds the return value to the variable x to produce y.

In the equivalent ARM code, we define two labels – *foo* and *main*. We assume that execution starts from the *main* label. We map x to $r1$, and set its value equal to 3. Then, we call the function *foo*. In it we set the value of register $r0$ to 2, and return by moving the value in the *lr* register to the PC. When the program returns, it begins execution at the subsequent line in the main function. The register $r0$ maintains its value equal to 2 across functions. We add the value in $r1$ to the value in $r0$ to produce the value for y. It is saved in $r2$.

Nowadays, there is a simpler method is used to return from a function. We can use the *bx* instruction that jumps to an address contained in a register (semantics shown in Figure 4.11).

Semantics	Example	Explanation
bx *reg*	bx r2	(1) Jump unconditionally to the address contained in register, r2

Table 4.11: Semantics of the *bx* instruction

We can simplify the assembly code in Example 46 as follows.

ARM assembly
```
foo:
    mov r0, #2
    bx lr

main:
    mov r1, #3       /* x = 3 */
    bl foo           /* invoke foo */
                     /*  y = x + foo() */
    add r2, r0, r1
```

4.2.8 Conditional Instructions

Now, that we have a fairly good idea of basic branch instructions, let us elaborate some special features of ARM assembly. These features help make the process of coding very efficient. Let us consider the instructions *beq* and *bne* again. We note that they are variants of the basic *b* instruction. They are distinguished by their suffixes – *eq* and *ne*. The former denotes equality, and the latter denotes inequality. These suffixes are known as *condition codes*

ARM Condition Codes

Let us first consider the list of *condition codes* shown in Table 4.12. There are 16 condition codes in ARM. Each condition code has a unique number, and suffix. For example, the condition code with suffix *eq* has a number equal to 0. Every condition code is associated with a unique condition. For example, *eq* is associated with equality. To test if the condition holds, the ARM

Number	Suffix	Meaning	Flag State
0	eq	equal	Z = 1
1	ne	not equal	Z = 0
2	cs/hs	carry set/ unsigned higher or equal	C = 1
3	cc/lo	carry clear/ unsigned lower	C = 0
4	mi	negative/ minus	N = 1
5	pl	positive or zero/ plus	N = 0
6	vs	overflow	V = 1
7	vc	no overflow	V = 0
8	hi	unsigned higher	$(C = 1) \wedge (Z = 0)$
9	ls	unsigned lower or equal	$(C = 0) \vee (Z = 1)$
10	ge	signed greater than or equal	N = 0
11	lt	signed less than	N = 1
12	gt	signed greater than	$(Z = 0) \wedge (N = 0)$
13	le	signed less than or equal	$(Z = 1) \vee (N = 1)$
14	al	always	
15	–	reserved	

Table 4.12: Condition codes

processor takes a look at the CPSR flags. The last column in Table 4.12 shows the values of the flags that need to be set for the condition to hold.

The *eq* and *ne* conditions can be tested by considering the Z(zero) flag alone. The expectation is that an earlier *cmp* or *subs* instruction would have set these flags. If the comparison resulted in an equality, then the Z flag would be set to 1.

As described in Section 4.2.3, if a subtraction of unsigned numbers leads to a borrow, then the carry flag is set to 0. This condition is also known as an unsigned underflow. If there is no borrow, then the carry flag is set to 1. Consequently, if the comparison between unsigned numbers concludes that the first number is greater than or equal to the second number, then the C(carry flag) needs to be set to 1. Likewise, if the carry flag is set to 0, then we can say that the first operand is smaller than the second operand (unsigned comparison). These two conditions are captured by the *hs* and *lo* condition codes respectively.

The next four condition codes check if a number is positive or negative, and if there has been an overflow. These conditions can be trivially evaluated by considering the values of N(negative) and V(overflow) flags respectively. *hi* denotes unsigned higher. In this case, we need to additionally test the Z flag. Likewise for *ls* (unsigned lower or equal), we need to test the Z flag, along with the C flag.

ARM has four condition codes for signed numbers – $ge(\geq)$, $le(\leq)$, $gt(>)$, and $lt(<)$. The *ge* condition code simply tests the N flag. It should be equal to 0. This means that a preceding *cmp* or *subs* instruction has subtracted two numbers, where the first operand was greater than or equal to the second operand. For the *gt* instruction, we need to consider the Z flag also. In a similar manner, the less than condition codes – *lt* and *le* – work. The conditions for the flags are given in Table 4.12.

Note that for signed numbers, we have not considered the possibility of an overflow in Table 4.12. Theorem 2.3.4.1 outlines the precise conditions for detecting an overflow. We leave the process of augmenting the conditions to consider overflow as an exercise for the reader. Lastly, the *al*(always) condition code means that the instruction is not associated with any condition. It executes according to its default specification. Hence, it is not required to explicitly specify the *al* condition since it is the default.

Conditional Variants of Normal Instructions

Condition codes are not just restricted to branches. We can use condition codes with normal instructions such as *add* and *sub* also. For example, the instruction *addeq* performs an addition if the Z flag in the $flags$ register is set to true. It means that the last time that the flags were set (most likely by a *cmp* instruction), the instruction must have concluded an equality. However, if the last comparison instruction concluded that its operands are unequal, then the ARM processor treats the *addeq* instruction as a *nop* instruction (no operation). We shall see in Chapter 10 that by using such conditional instructions, we can increase the performance of an advanced processor. Let us consider an example that uses the *addeq* instruction.

Example 47
Write a program in ARM assembly to count the number of 1s in a 32-bit number stored in $r1$*. Save the result in* $r4$*.*
Answer:

```
mov r2, #1 /* idx = 1 */
mov r4, #0 /* count = 0 */

/* start the iterations */
.loop:
        /* extract the LSB and compare */
        and r3, r1, #1
        cmp r3, #1

        /* increment the counter */
        addeq r4, r4, #1

        /* prepare for the next iteration */
        mov r1, r1, lsr #1
        add r2, r2, #1

        /* loop condition */
        cmp r2, #32
        ble .loop
```

4.2.9 Load-Store Instructions

Simple Load-Store Instructions

The simplest load and store instructions are *ldr* and *str* respectively. Here, is an example.

```
ldr r1, [r0]
```

This instruction directs the processor to load the value in register $r1$, from the memory location stored in $r0$, as shown in Figure 4.2.

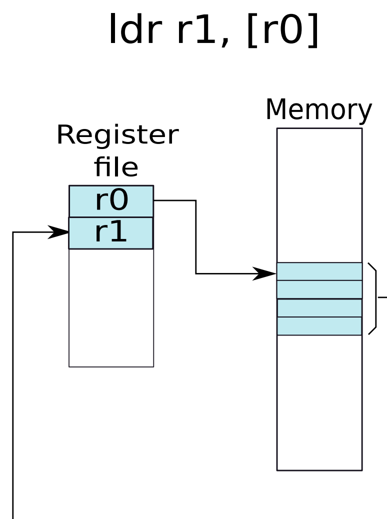

Figure 4.2: The *ldr* instruction

Note that in this case, $r0$, contains the starting address of the data in memory. The *ldr* instructions loads 4 bytes in a register. If the value contained in $r0$ is v, then we need to fetch the bytes from v to $v + 3$. These 32 bits (4 bytes), are brought from memory and saved in register $r1$.

The *str* instruction performs the reverse process. It reads the value in a register and saves it in a memory location. An example is shown in Figure 4.3. Here $r0$ is known as the base register.

```
str r1, [r0]
```

Load-Store Instructions with an Offset

We can specify load and store instructions with a base register, and an optional offset. Let us consider:

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

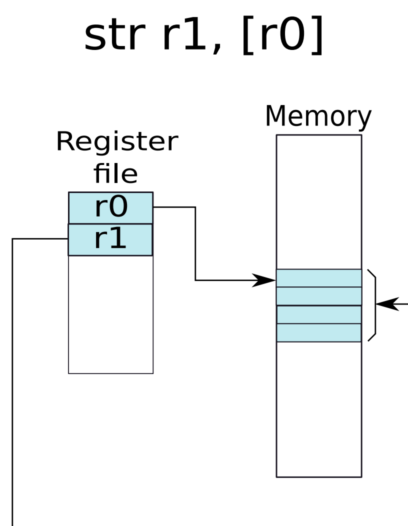

Figure 4.3: The *str* instruction

Here, the memory address is equal to the value in $r0$ plus 4. It is possible to specify a register in place of an immediate operand.

```
ldr r1, [r0, r2]
```

The memory address is equal to $r0 + r2$. In this expression, $r0$ and $r2$ refer to the values stored in them. We can alternatively state the operation in this program as: $r1 \leftarrow [r0 + r2]$ (see the register transfer notation defined in Section 3.2.5).

Semantics	Example	Explanation	Addressing Mode
ldr *reg*, [*reg*]	ldr r1, [r0]	$r1 \leftarrow [r0]$	register-indirect
ldr *reg*, [*reg*, *imm*]	ldr r1, [r0, #4]	$r1 \leftarrow [r0 + 4]$	base-offset
ldr *reg*, [*reg*, *reg*]	ldr r1, [r0, r2]	$r1 \leftarrow [r0 + r2]$	base-index
ldr *reg*, [*reg*, *reg*, shift *imm*]	ldr r1, [r0, r2, lsl #2]	$r1 \leftarrow [r0 + r2 \ll 2]$	base-scaled-index
str *reg*, [*reg*]	str r1, [r0]	$[r0] \leftarrow r1$	register-indirect
str *reg*, [*reg*, *imm*]	str r1, [r0, #4]	$[r0 + 4] \leftarrow r1$	base-offset
str *reg*, [*reg*, *reg*]	str r1, [r0, r2]	$[r0 + r2] \leftarrow r1$	base-index
str *reg*, [*reg*, *reg*, shift *imm*]	str r1, [r0, r2, lsl #2]	$[r0 + r2 \ll 2] \leftarrow r1$	base-scaled-index

Table 4.13: Load and store instruction semantics

Table 4.13 shows the semantics of different types of load store instructions. The third column shows the *addressing mode*. The register $r2$ in this case is known as the index register because it contains a value that is added to the base register, and this value can be used as the index of an array (see Section 4.3.1). Note that some authors call the base-offset mode as also the *displacement* addressing mode.

Load-Store instructions for Bytes and Half-Words

The *ldr* and *str* instructions load/store 4 bytes of data. However, it is possible to also load and store 1 and 2 bytes of data. 2 bytes is also known as a half-word, where a word is equal to 4 bytes.

Semantics	Example	Explanation
ldrb *reg*, [*reg*, *imm*]	ldrb r1, [r0, #2]	$r1 \leftarrow [r0 + 2]$ (1 unsigned byte)
ldrh *reg*, [*reg*, *imm*]	ldrh r1, [r0, #2]	$r1 \leftarrow [r0 + 2]$ (2 unsigned bytes)
ldrsb *reg*, [*reg*, *imm*]	ldrsb r1, [r0, #2]	$r1 \leftarrow [r0 + 2]$ (1 signed byte)
ldrsh *reg*, [*reg*, *imm*]	ldrsh r1, [r0, #2]	$r1 \leftarrow [r0 + 2]$ (2 signed bytes)
strb *reg*, [*reg*, *imm*]	strb r1, [r0, #2]	$[r0 + 2] \leftarrow r1$ (1 unsigned byte)
strh *reg*, [*reg*, *imm*]	strh r1, [r0, #2]	$[r0 + 2] \leftarrow r1$ (2 unsigned bytes)

Table 4.14: Load and store instructions for bytes and half-words in the base-offset addressing mode

Table 4.14 shows the load and store instructions for bytes and half words using the base-offset addressing mode. *ldrb* loads an unsigned byte to a register. It places the byte in the least significant 8 bits. The rest of the 24 bits are set to 0. *ldrh* similarly loads an unsigned half-word (16 bits). *ldrsb*, and *ldrsh* load a signed byte and half-word respectively. They extend the sign of the operand (see Section 2.3.4) to make it fit in 32 bits. This is done by replicating the MSB. *strb* and *strh* store an unsigned byte in memory. Note that unlike loads, there are no ARM instructions to extend the sign of the operand while saving it in memory.

4.3 Advanced Features

We are in a good point to take a look at some of the advanced features in the ARM instruction set. Up till now, we have taken a look at basic instructions that allow us to implement simple data types in a high level language such as C or Java. We can translate simple programs that contain integers into assembly code, compute the results of mathematical functions, load and store values from memory. However, there are other high level features such as functions, arrays, and structures that are present in high level languages. They shall require special support at the assembly level for creating efficient implementations.

By no means has the process of programming language development stopped. We expect that over the next few decades, there will be many new kinds of programming languages. They will make the process of programming easier for more programmers, and it should be easier to leverage novel features of futuristic hardware. This would require extra instructions and support at the level of assembly programs. This is thus an evolving field, and deserves a thorough study.

4.3.1 Arrays

Array Specific Features

Note that the starting memory location of entry i is equal to the base address of the array plus $4i$ in an array with word (4 byte) sized elements. In a high level language, the programmer always specifies the index in an array, and relies on the compiler to multiply the index by 4. ARM assembly provides nice features to multiply i by 4 by using the *lsl* instruction. This feature can be embedded in load-store instructions.

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

In this case the base address is stored in register, $r1$, and the offset is equal to $r2 << 2 = 4 * r2$. The advantage here is that we do not need a separate instruction to multiply the index by 4. We have already seen this optimisation in Section 4.2.2. However, there are other optimisations that can make our life easier. Let us consider array accesses in a loop as shown in Example 48.

Example 48 *Convert the following C program to a program to ARM assembly. Assume that the base address of the array is stored in* $r0$*.*

C

```
void addNumbers(int a[100]) {
    int idx;
    int sum = 0;
    for (idx = 0; idx < 100; idx++){
        sum = sum + a[idx];
    }
}
```

Answer:

ARM assembly

```
1  /* base address of array a in r0 */
2  mov r1, #0  /* sum = 0 */
3  mov r2, #0  /* idx = 0 */
4
5  .loop:
6      ldr r3, [r0, r2, lsl #2]
7      add r2, r2, #1  /* idx ++  */
8      add r1, r1, r3  /* sum += a[idx] */
9      cmp r2, #100    /* loop condition */
10     bne .loop
```

There is a scope for added efficiency here. We note that Lines 6 and 7 form a standard pattern. Line 6 reads the array entry, and Line 7 increments the index. Almost all sequential

array accesses follow a similar pattern. Hence, it makes sense to have one instruction that simplifies this process.

The ARM architecture adds two extra addressing modes for the load and store instructions to achieve this. They are called *pre-indexed* and *post-indexed* with auto-update. In the pre-indexed addressing mode (with auto-update), the base address is updated first, and then the effective memory address is computed. In a post-indexed scheme, the base address is updated after the effective address is computed.

The pre-indexed addressing mode with auto-update is implemented by adding a '!' sign after the address.

Examples of the pre-indexed addressing mode

```
ldr r3, [r0, #4]!         /*  r3 = [r0+4]; r0 = r0 + 4*/
ldr r3, [r0, r1, lsl #2]! /* r3 = [r0 + r1 << 2];
                             r0 = r0 + r1 << 2;  */
```

The post-indexed addressing mode is implemented by encapsulating the base address within '[' and ']', and writing the offset arguments separated by commas after it.

Examples of the post-indexed addressing mode

```
ldr r3, [r0], #4          /* r3 = [r0], r0 = r0 + 4 */
ldr r3, [r0], r1, lsl #2  /* r3 = [r0], r0 = r0 + r1 << 2 */
```

Let us now see, how we can slightly make our $addNumbers$ slightly more intuitive. The modified ARM code is shown in Example 49.

Example 49

Convert the assembly code shown in Example 48 to use the post indexed addressing mode.

Answer:

ARM assembly

```
1 /* base address of array a in r0 */
2     mov r1, #0        /* sum = 0 */
3     add r4, r0, #400  /* address of a[100]*/
4 .loop:
5         ldr r3, [r0], #4
6     add r1, r1, r3    /* sum += a[idx] */
7     cmp r0, r4
8     bne .loop
```

We have eliminated the index variable saved in $r2$. It is not required anymore. We directly update the base address in Line 5. For the loop exit condition, we compute the first address beyond the end of the array in Line 3. We compare the base address with this illegal address in Line 7, and then if they are unequal we keep iterating.

Example 48 contains 5 lines in the loop, whereas the code in Example 49 contains 4 lines in the loop. We have thus shown that it is possible to reduce the code size (of the loop) by 20% using post-indexed addressing, and increase performance too since most cores do not impose additional time overheads when auto-update addressing modes are used.

Structures

Implementing structures is very similar to implementing arrays. Let us look at a typical structure in C.

```
struct Container {
    int a;
    int b;
    char c;
    short int d;
    int e;
};
```

We can treat each structure as an array. Consequently, a structure will have a base address and each element of the structure will have an offset. Unlike an array, different elements in a structure can have different sizes, and thus they are not constrained to start with offsets that are multiples of the word size.

Type	Element	Offset
int	a	0
int	b	4
char	c	8
short int	d	10
int	e	12

Table 4.15: Elements in the structure and their offsets

Table 4.15 shows the offsets for different elements within a structure (as generated by the GNU ARM compiler). We need to note that compilers for the ARM architecture impose additional constraints. They pad variable addresses, and align them with 2 byte or 4 byte boundaries as shown in Table 4.15 The rules for variable alignment are described in detail in the ARM architecture manual [arm, 2000]. In a similar fashion it is possible to implement more high level data structures such as unions and classes. The interested reader is referred to a book on compilers.

4.3.2 Functions

Let us now use two sophisticated ARM instructions for spilling and restoring registers in the stack. They can be used to implement both caller saved and callee saved functions.

Instructions for Spilling and Restoring Registers

Let us now describe two instructions to use the stack for saving and restoring a set of registers – $ldmfd$ and $stmfd$. These registers load and store multiple registers in a memory region such as the stack. For brevity, we do not consider generic memory regions in this book. We limit our discussion to the stack. $ldmfd$ and $stmfd$ instructions take a base register (e.g., stack pointer),

and set of registers as arguments. They load or store the set of registers in the memory region pointed to by the base register. Note that the order of the registers does not matter. The registers are always rearranged in ascending order.

Let us consider an example using the store instruction, $stmfd$.

```
stmfd sp!, {r2,r3,r1,r4}
```

Instruction	Semantics
ldmfd sp!, {list of registers }	Pop the stack and assign values to registers in ascending order. Update the value of sp.
stmfd sp!, {list of registers }	Push the registers on the stack in descending order. Update the value of sp.

Table 4.16: Semantics of the $ldmfd$ and $stmfd$ instructions

The $stmfd$ instruction assumes a downward growing stack, and it also assumes that the stack pointer points to the starting address of the value at the top of the stack. Recall that the top of the stack in a downward growing stack is defined as the starting address of the last value pushed on the stack. In this case the registers are processed in ascending order – $r1$, $r2$, $r3$, $r4$. Secondly memory addresses are also accessed in ascending order. Consequently $r1$ will be saved in $sp - 16$, $r2$ in $sp - 12$, $r3$ in $sp - 8$, and $r4$ in $sp - 4$. Alternatively, we can explain this instruction by observing that registers are pushed into the stack in descending order. We use the '!' suffix with the base address register to instruct the processor to update the value of the stack pointer after the execution of the instruction. In this case, we set sp equal to $sp - 16$.

There is a variant of this instruction that does not set the stack pointer to the starting address of the memory region used to save registers. An example with this variant is:

```
stmfd sp, {r2,r3,r1,r4}
```

Note that this variant is rarely used in practice, especially when the base register is sp.

Similarly, the $ldmfd$ instruction loads a set of values starting at the stack pointer, and then updates the stack pointer. Akin to the $stmfd$ instruction, we use the '!' suffix to use the base register auto update feature.

```
ldmfd sp!, {r2,r3,r1,r4}
```

For example, in this case we set $r1 = [sp]$, $r2 = [sp+4]$, $r3 = [sp+8]$, and $r4 = [sp+12]$. In other words, we iteratively pop the stack and assign the values to registers in ascending order. The $ldmfd$ instruction also has a variant that does not update the base register. We simply need to delete the '!' suffix after the base register.

```
ldmfd sp, {r2,r3,r1,r4}
```

The semantics of these instructions are shown in Table 4.16.

Let us conclude this section with an example. We show a recursive power function in C that takes two arguments x and n, and computes x^n.

Example 50

Write a function in C and implement it in ARM assembly to compute x^n, where x and n are natural numbers. Assume that x is passed through $r0$, n through $r1$, and the return value is passed back to the original program via $r0$. ***Answer:***

C
```c
int power(int x, int n) {
	if (n == 0)
		return 1;
	int y = x * power(x, n-1);
	return y;
}
```

When we compile this function to ARM assembly, we get:

ARM assembly
```
1  power:
2      cmp r1, #0            /* compare n with 0  */
3      moveq r0, #1          /* return 1 */
4      bxeq pc, lr           /* return */
5
6      stmfd sp!, {r4, lr}   /* save r4 and lr */
7      mov r4, r0            /* save x in r4 */
8      sub r1, r1, #1        /* n = n - 1 */
9      bl power              /* recursively call power */
10     mul r0, r4, r0        /* power(x,n) = x * power(x,n-1) */
11     ldmfd sp!, {r4, pc}   /* restore r4 and return */
```

We first compare n with 0. If n is equal to 0, then we need to return 1 (Line 3). We subsequently, return from the function. Note the use of the instruction moveq here.

However, if $n \neq 0$, then we need to make a recursive function call to evaluate x^{n-1}. We start out by saving register $r4$, and the return address (lr) on the stack in Line 6 using the $stmfd$ instruction. We save the value of $r0$ in $r4$ because it will get overwritten by the recursive call to the power function. Subsequently, we decrement $r1$ that contains the value of n, and then we call the power function recursively in Line 10. The result of the power function is assumed to be present in $r0$. We multiply this result with the value of x (stored in $r4$) in Line 10.

We simultaneously do two operations in Line 11. We load the value of $r4$, and pc from the stack. We first read the first operand, $r4$, which was saved on the stack by the corresponding $stmfd$ instruction in Line 6. The second operand saved on the stack was the return address. We read this value and save it in pc. Effectively, we are executing the

instruction mov pc, lr*, and we are thus returning from the function. Hence, after executing Line 11, we start executing instructions from the return address of the function.*

The *ldm* and *stm* instructions can also assume an upward growing stack. The interested reader can refer to the ARM manual [arm, 2000] for a thorough explanation.

4.4 Encoding the Instruction Set

Let us now see how to convert ARM assembly instructions to a sequence of 0s and 1s. Each ARM instruction is represented using 32 bits. We need to encode the instruction type, values of conditional fields, register numbers, and immediate operands using these 32 bits only.

Let us take a look at the generic format of ARM instructions. For every instruction we need to initially encode at least two pieces of information – condition codes (see Table 4.12), and the format of the instruction (data processing, branch, load/store, or others). Table 4.12 defines 15 conditions on each instruction. It will take 4 bits to represent this information.

Important Point 6
To uniquely encode a set of n elements, we need at least $\lceil log_2(n) \rceil$ bits. We can assign each element a number between 0 and $n-1$. We can represent these numbers in the binary format. The number of bits required is equal to the number of bits needed to represent the largest number, $n-1$. If we have $log_2(n)$ bits, then the largest number that we can represent is $2^{log_2(n)}-1 = n-1$. However, $log_2(n)$ might be a fraction. Hence, we need to use $\lceil log_2(n) \rceil$ bits.

ARM has four types of instructions – data processing (add/ subtract/ multiply/ compare), load/store, branch, and miscellaneous. We need 2 bits to represent this information. These bits determine the type of the instruction. Figure 4.4 shows the generic format for instructions in ARM.

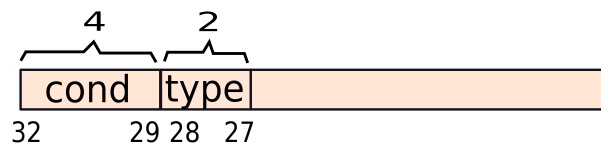

Figure 4.4: Generic format of an ARM instruction

4.4.1 Data Processing Instructions

The type field is equal to 00 for data processing instructions. The rest of the 26 bits need to contain the instruction type, special conditions, and registers. Figure 4.5 shows the format for

data processing instructions.

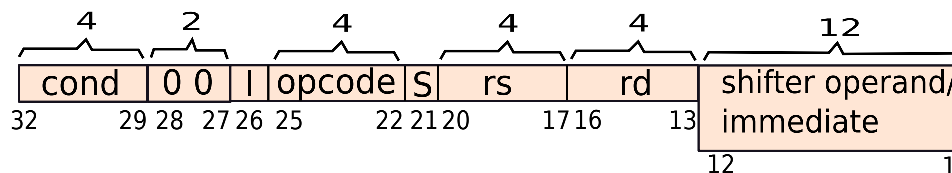

Figure 4.5: Format of the data processing instruction

The 26th bit is called the I (immediate) bit. It is similar to the I bit in *SimpleRisc* . If it is set to 1, then the second operand is an immediate, otherwise, it is a register. Since ARM has 16 data processing instructions, we require 4 bits to represent them. This information is saved in bits 22-25. The 21st bit saves the S bit. If it is turned on, then the instruction will set the CPSR (see Section 4.2.4).

The rest of the 20 bits save the input and output operands. Since ARM has 16 registers, we require 4 bits to encode a register. Bits 17-20 save the identifier of the first input operand (rs), which needs to be a register. Bits 13-16 save the identifier of the destination register (rd).

Bits 1-12 are used to save the immediate value or the shifter operand. Let us see how to make best use of these 12 bits.

Encoding Immediate Values

ARM supports 32-bit immediate values. However, we observe that we have only 12 bits to encode them. Hence, we cannot possibly encode all the 2^{32} possible values. We need to choose a meaningful subset of them. The idea is to encode a subset of 32-bit values using 12 bits. The hardware is expected to decode these 12 bits, and expand them to 32 bits while processing the instruction.

Now, 12 bits is a rather unwieldy value. Neither is it 1 byte nor is it 2 bytes. Hence, it was necessary to come up with a very ingenious solution. The idea is to split the 12 bits into two parts – a 4-bit constant (rot), and an 8 bit payload ($payload$) (see Figure 4.6).

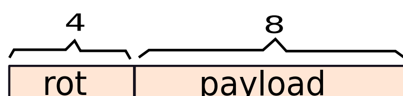

Figure 4.6: Format of the immediate

Let the actual number that is encoded in these 12 bits be n. We have:

$$n = payload\ ror\ (2 \times rot)$$

The actual number n is obtained by right rotating the payload by 2 times the value in the rot field. Let us now try to understand the logic of doing so.

The final number n is a 32-bit value. A naive solution would have been to use the 12 bits to specify the least significant bits of n. The higher order bits could be 0. However, programmers tend to access data and memory in terms of bytes. Hence, 1.5 bytes is of no use to us. A better solution is to have a 1 byte payload and place it in any location in the 32-bit field. The rest of the 4 bits are used for this purpose. They can encode a number from 0 to 15. The ARM processor doubles this value to consider all even numbers between 0 and 30. It right rotates the payload by this amount. The advantage of doing so is that it is possible to encode a wider set of numbers. For all of these numbers, there are 8 bits that correspond to the payload, and the rest of the 24 bits are all zeros. The *rot* bits just determine which 8 bits in a 32-bit field are occupied by the payload.

Let us consider a set of examples.

Example 51
Encode the decimal number 42.
Answer: *42 in the hex format is 0x2A, or alternatively 0x00 00 00 2A. There is no right rotation involved. Hence, the immediate field is 0x02A.*

Example 52
Encode the number 0x2A 00 00 00.
Answer: *This number is obtained by right rotating 0x2A by 8 places. Note that we need to right rotate by 4 places to move a hex digit by one position. We need to now divide 8 by 2, to get 4. Thus, the encoded format for this number is 0x42A.*

Example 53
Encode 0x 00 00 2A 00.
Answer: *The first step is to count the number of right rotations. We observe that the number 0x2A has been rotated to the right by 24 positions. We now proceed to divide 24 by 2 to obtain 12. Thus, the encoded format of the number is 0xC2A.*

Example 54
Encode the number 0x 00 02 DC 00 as an ARM immediate.
Answer: *The first part is to figure out the payload. The payload is – 10 1101 11 – in binary. This is equal to 0xB7. The next step is to figure out the rotation. Let us simplify the task by observing that right rotating by n places is the same as left rotating by $32-n$ places. Let us concentrate on 0xC00. This is equal to* 110000000000 *in binary. The rightmost 1 is*

now at the 11th position. It has moved 10 places from the 1st position. Thus the number has been rotated to the left by 10 places. It has been rotated to the right by 22 places. $22/2 = 11(0xB)$. *Hence, the encoded number is 0xBB7.*

The reader needs to understand that this encoding is supposed to be done by the assembler or the compiler. The user simply needs to only use values in her assembly code that can be encoded as an ARM immediate. For example, a number like -1 cannot be encoded as an ARM immediate. It is 0xFF FF FF FF. The payload is greater than 8 bits. Ideally, an instruction of the form: $add\ r1, r1, \# -1$ is wrong. Some assemblers will try to fix the problem by changing the instruction to $sub\ r1, r1, \#1$. However, all assemblers are not smart enough to figure this out. If the user wishes to uses a value that cannot be encoded in ARM's 12 bit format, then the user (or the program loader) needs to load it byte by byte in a register, and use the register as an operand.

Encoding the Shifter Operand

We have 12 bits to encode the shifter operand. Figure 4.7 shows the scheme for encoding it. A shifter operand is of the form: rt (lsl|lsr|asr|ror) (shift reg/ shift imm.)

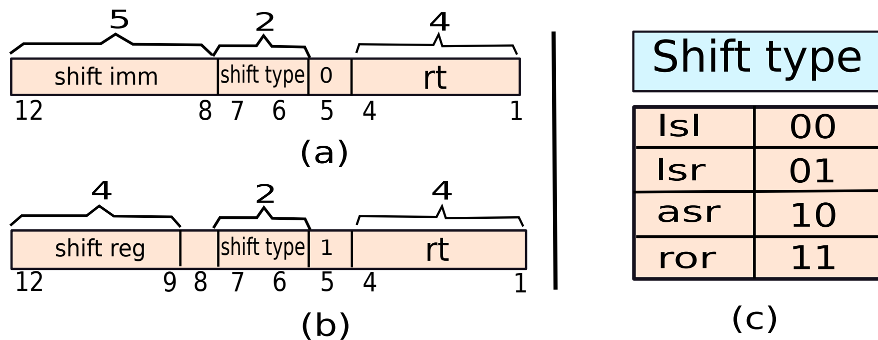

Figure 4.7: Format of the shifter operand

The first four bits (1-4) encode the id of the register rt. The next bit determines the nature of the shift argument (immediate or register). If it is 0 then the argument is an immediate, otherwise it is a register. Bits 6 and 7 specify the type of the shift (also see Figure 4.7(c)). For example, the type can be *lsl* (logical shift left). It can also be *lsr* (logic shift right), *asr* (arithmetic shift right), or *ror* (right rotate). If we are shifting by an immediate value, then bits 8-12 specify a 32-bit value called a *shift immediate*. Otherwise, if we are shifting by a value in a register, then bits 9-12 specify the id of the register.

Let us consider an instruction of the form: $add\ r3, r1, r2$. In this case, the second operand is $r2$. We can think of $r2$ as actually a shifter operand where it is being left shifted by 0. Hence, to encode we need to set the shift type to *lsl* (00), set the argument to immediate (0), and set

the shift immediate to 00000. We thus see that specifying a register as the second argument is easy. It is a special case of a shifter operand, and we just need to set bits 5-12 as 0.

4.4.2 Load-Store Instructions

A simple load or store instruction can be represented as : (ldr | str) rd, [rs, (immediate/shifter operand)]. We require additional syntax for pre and post-indexed addressing (see Section 4.3.1). The format for the encoding of load and store instructions is shown in Figure 4.8.

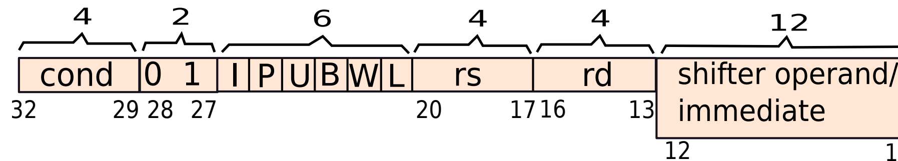

Figure 4.8: Format of the load/store instructions

The semantics of the bits I, P, U, B, W, and L is shown in Table 4.17. In this case, the I bit has reverse semantics as compared to the case of data processing instructions. If it is 1, then the last 12 bits represent a shifter operand, otherwise they represent an immediate value. P represents the advanced addressing mode – pre or post, and W determines if the advanced addressing mode is used or a simple addressing mode is used. We can either add the offset from the base register or we can subtract it from the base register. This is specified by the U bit. The B bit determines the granularity of the transfer – byte level or word level. Lastly, the L bit determines if the instruction is a load or a store.

Bit	Value	Semantics
I	0	last 12 bits represent an immediate value
	1	last 12 bits represent a shifter operand
P	0	post-indexed addressing
	1	pre-indexed addressing
U	0	subtract offset from base
	1	add offset to base
B	0	transfer word
	1	transfer byte
W	0	do not use pre or post indexed addressing
	1	use pre or post indexed addressing
L	0	store to memory
	1	load from memory

Table 4.17: Semantics of I, P, U, B, W, and L bits

These six bits $IPUBWL$ capture all the different variants of the load and store instructions. The rest of the format is the same as the data processing instruction other than the encoding of

immediates. Immediates in memory instructions do not follow the (rot+payload) format. The 12 bit immediate fields represents an unsigned number between 0 and 4095.

We thus observe that like *SimpleRisc* , the designers of the ARM instruction set have tried to stick to the same instruction format with minor variations..

Question 6 *What is the necessity for having the U bit?*
Answer: *Negative numbers such as -4 or -8 cannot be represented in ARM's 12 bit format for specifying offsets in memory instructions. However, we might need to use addresses with a negative displacement, especially when they are relative to the frame pointer or the stack pointer. The U bit allows us to represent an immediate such as -4 as +4. It additionally instructs the processor to subtract the displacement from the base register.*

4.4.3 Branch Instructions

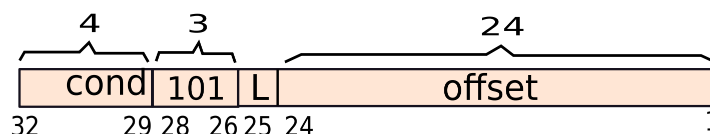

Figure 4.9: Format of the branch and branch-and-link instructions

Figure 4.9 shows the format of the branch (*b*) and the branch-and-link (*bl*) instructions. If the L(link) bit is equal to 1, then the instruction is *bl*, otherwise it is just *b*. The instruction contains a 24-bit signed offset. The ARM processor first shifts the offset by 2 bits. This is because each instruction is 32 bits or 4 bytes long, and additionally the hardware expects instructions to be stored at 4 byte boundaries. Therefore, the starting address of each instruction will contain two zeros in its two least significant positions. Hence, there is no necessity to waste two bits in the encoding for saving these two zeros. The next step is to extend the sign of this shifted offset to 32 bits. Lastly, the hardware computes the branch target by adding the shifted and sign-extended offset to the PC plus 8 bytes.

The interesting thing to note is that we are adding the sign-extended shifted offset to PC+8, not the PC. We shall see in Chapter 10 that the reason for doing this is to simplify the hardware. The format for branches is different from the format used to encode data transfer and data processing instructions. This is because more bits have used to encode the displacement. We had followed a similar approach in *SimpleRisc* also. However, we need to note that having a new format is not a very bad thing if it is simple as is the case for a branch.

4.5 Summary and Further Reading

4.5.1 Summary

Summary 4

1. *The ARM ISA is a simple 32-bit RISC ISA.*
   (a) *It uses 16 registers* $r0 \ldots r15$.
   (b) *The return address register is known as lr (link register), and it is* $r14$.
   (c) *The PC is visible to the programmer. It is register* $r15$.
   (d) *All the instructions are encoded using 32 bits.*
2. *Data processing instructions accept register operands, and at most one immediate operand. They are 3-address instructions.*
3. *ARM has a set of compare instructions that can set flags in the CPSR register. Additionally, it is possible to instruct a standard data processing instruction to set the CPSR flags by adding the suffix 's' to it.*
4. *ARM supports conditional instructions that either execute or not depending upon the values of the CPSR flags. They can be created by appending a condition code to a regular data processing or branch instruction. There are 15 such condition codes. Examples of some condition codes are: gt (greater than), and eq (equal).*
5. *ARM has two variants of branch instructions.*
   (a) *It has simple branch instructions that branch to another instruction.*
   (b) *It has branch-and-link instructions that additionally save the return address in the link register* lr.
6. *ARM supports both the base-index and base-offset addressing modes for load and store instructions. It has additional support for shifting the index register by treating it as a shifter operand.*
7. *ARM supports complex addressing modes such as pre-indexed and post-indexed addressing. These addressing modes update the base register.*
8. *ARM also has support for loading and storing bytes and half-words (2 bytes).*
9. *The instruction set encoding for data processing instructions is as follows:*
   (a) *Condition code (4 bits)*
   (b) *Instruction type (2 bits)*
   (c) *Second operand: immediate or register (1 bit)*

(d) Opcode (4 bits)

(e) S bit (should the CPSR flags be set) (1 bit)

(f) Source register1 (4 bits)

(g) Destination register (4 bits)

(h) Immediate or shifter operand (12 bits)

10. The data transfer instructions do not have the S bit. They instead have extra bits to encode the type of load/store instructions, and the addressing mode.

11. The branch instructions have an L bit to specify if the return address needs to be saved or not. They use PC-relative addressing and have a 24-bit signed offset. Like SimpleRisc , the hardware assumes that instructions are aligned to 4 byte boundaries, and treats this offset as a distance in terms of memory words. It thus left shifts the offset by 2 positions.

4.5.2 Further Reading

We have presented an overview of the major features of ARM's assembly language. The reader can refer to ARM's assembly language manual [arm, 2000] for more details.

We have deliberately left out some advanced features. A subset of ARM cores support Thumb-1 and Thumb-2 instructions. These instructions are based on a subset of general purpose instructions and have implicit operands. They are used to decrease the size of compiled code. Some ARM processors have extensive support for floating point instructions (VFP instruction set), and SIMD instructions (execute an instruction on multiple integers/floating point numbers in one go). However, we have not discussed these extensions for the sake of brevity. Some other sophisticated features of ARM processors are security extensions that prevent malicious programs or users from stealing data. Since 2013 ARM processors (conforming to the ARMv8-A architecture) have started using a new 64-bit ARM ISA called A64. The reader can refer to the books by Joseph Yiu [Yiu, 2011, Yiu, 2009], William Hohl [Hohl, 2009], and J. R. Gibson [Gibson, 2011] for a detailed discussion on the ARM instruction set and its latest extensions. Needless to say the reader can always find up to date documentation at ARM's web site `http://www.arm.com`.

Exercises

Basic ARM Instructions

Ex. 1 — Translate the following code in C to the ARM instruction set using a minimum number of instructions. Assume the variables a, b, c, d and e are 32-bit integers and stored in $r0$, $r1$, $r2$, $r3$ and $r4$ respectively.

(a) `a=a+b+c+d+e;`

(b) `a=b+c;`
`d=a+b;`

(c) `a=b+c+d;`
`a=a+a;`

(d) `a=2*a+b+c+d;`

(e) `a=b+c+d;`
`a=3*a;`

Ex. 2 — Translate the following pieces of code from the ARM assembly language to a high level language. Assume that the variables a, b, c, d and e (containing integers) are stored in the registers $r0$, $r1$, $r2$, $r3$ and $r4$ respectively.

(a)
```
add r0, r0, r1
add r0, r0, r2
add r0, r0, r3
```

(b)
```
orr r0, r0, r1, lsl #1
and r1, r0, r1, lsr #1
```

(c)
```
add r0, r1, r2
rsb r1, r0, r2
```

(d)
```
add r0, r1, r2
add r0, r3, r4
add r0, r0, r1
```

(e)
```
mov r0 #1, lsl #3
mov r0, r0, lsr #1
```

Ex. 3 — Answer the following:

(a) Write the smallest possible ARM assembly program to load the constant 0xEFFFFFF2 into register $r0$.

(*b) Write the smallest possible ARM assembly program to load the constant 0xFFFD67FF into register $r0$.

* **Ex. 4** — Using valid ARM assembly instructions, load the constant, 0xFE0D9FFF, into register $r0$. Try do to it with a minimum number of instructions. DO NOT use pseudo-instructions or assembler directives.

Ex. 5 — Can you give a generic set of ARM instructions or a methodology using which you can load any 32-bit immediate value into a register? Try to minimise the number of instructions.

Ex. 6 — Convert the following C program to ARM assembly. Store the integer, i, in register $r0$. Assume that the starting address of array a is saved in register $r1$, and the starting address of array b is saved in register $r2$.

```
int i;
int b[500];
int a[500];
for(i=0; i < 500; i++) {
    b[i] = a[a[i]];
}
```

** **Ex. 7** — Consider the instruction, *mov lr, pc*. Why does this instruction add 8 to the PC, and use that value to set the value of *lr*? When is this behaviour helpful?

Assembly Language Programming

- For all the questions below, assume that two specialised functions, *__div* and *__mod*, are available. The *__div* function divides the contents of $r1$ by the contents of $r2$, and saves the result in $r0$. Similarly, the *__mod* function is used to divide $r1$ by $r2$, and save the remainder in $r0$. Note that in this case both the functions perform integer division.

Ex. 8 — Write an ARM assembly language program to compute the 2's complement of a number stored in $r0$.

Ex. 9 — Write an ARM assembly language program that subtracts two 64-bit integers stored in four registers.
Assumptions:

- Assume that you are subtracting $A - B$
- A is stored in register, $r4$ and $r5$. The MSB is in $r4$, and the LSB is in $r5$.
- B is stored in register, $r6$ and $r7$. The MSB is in $r6$, and the LSB is in $r7$.
- Place the final result in $r8$(MSB), and $r9$(LSB).

Ex. 10 — Write an assembly program to add two 96-bit numbers A and B using the minimum number of instructions. A is stored in three registers $r2$, $r3$ and $r4$ with the higher byte in $r2$ and the lower byte in $r4$. B is stored in registers $r5$, $r6$ and $r7$ with the higher byte in $r5$ and the lower byte in $r7$. Place the final result in $r8$(higher byte), $r9$ and $r10$(lower byte).

Ex. 11 — Write an ARM assembly instruction code to count the number of 1's in a 32-bit number.

Ex. 12 — Given a 32-bit integer in $r3$, write an ARM assembly program to count the number of 1 to 0 transitions in it.

* **Ex. 13** — Write an ARM assembly program that checks if a 32-bit number is a palindrome. Assume that the input is available in $r3$. The program should set $r4$ to 1 if it is a palindrome, otherwise $r4$ should have 0. A palindrome is a number which is the same when read from both sides. For example, 1001 is a 4-bit palindrome.

Ex. 14 — Design an ARM Assembly Language program that will examine a 32-bit value stored in $r1$ and count the number of contiguous sequences of 1s. For example, the value:

01110001000111101100011100011111

contains six sequences of 1s. Write the final value in register $r2$. Use conditional instructions as much as possible.

** **Ex. 15** — In some cases, we can rotate an integer to the right by n positions (less than or equal to 31) so that we obtain the same number. For example: an 8-bit number 01010101 can be right rotated by 2, 4, or 6 places to obtain the same number. Write an ARM assembly program to *efficiently* count the number of ways we can rotate a number to the right such that the result is equal to the original number.

Ex. 16 — Write an ARM assembly program to load and store an integer from memory, where the memory saves it in the big endian format.

Ex. 17 — Write an ARM assembly program to find out if a number is prime using a recursive algorithm.

* **Ex. 18** — Suppose you decide to take your ARM device to some place with a high amount of radiation, which can cause some bits to flip, and consequently corrupt data. Hence, you decide to store a single bit checksum, which stores the parity of all the other bits, at the least significant position of the number (essentially you can now store only 31 bits of data in a register). Write an ARM assembly program, which adds two numbers taking care of the checksum. Assume that no bits flip while the program is running.

* **Ex. 19** — Let us encode a 16-bit number by using 2 bits to represent 1 bit. We shall represent logical 0 by 01, and logical 1 by 10. Now let us assume that a 16-bit number is encoded and stored in a 32-bit register $r3$. Write a program in ARM assembly to convert it back into a 16-bit number, and save the result in $r4$. Note that 00 and 11 are invalid inputs and indicate an error. The program should set $r5$ to 1 in case of an error; otherwise, $r5$ should be 0.

** **Ex. 20** — Write an ARM assembly program to convert a 32-bit number to its 12 bit immediate form, if possible, with first 4 bits for rotation and next 8 bits for the payload. If the conversion is possible, set $r4$ to 1 and store the result in $r5$, otherwise, $r4$ should be set to 0. Assume that the input number is available in register $r3$.

** **Ex. 21** — Suppose you are given a 32-bit binary number. You are told that the number has exactly one bit equal to 1; the rest of the bits are 0. Provide a fast algorithm to find the location of that bit. Implement the algorithm in ARM assembly. Assume the input to be available in $r9$. Store the result in $r10$.

*** **Ex. 22** — Write an ARM assembly language program to find the greatest common divisor of two binary numbers u and v. Assume the two inputs (positive integers) to be available in $r3$ and $r4$. Store the result in $r5$. [HINT: The gcd of two even numbers u and v is $2 * gcd(u/2, v/2)$]

ARM Instruction Encoding

Ex. 23 — How are immediate values encoded in the ARM ISA?

Ex. 24 — Encode the following ARM instructions. Find the opcodes for instructions from the ARM architecture manual [arm, 2000].

i) add r3, r1, r2
ii) ldr r1, [r0, r2]
iii) str r0, [r1, r2, lsl #2]

Design Problems

Ex. 25 — Run your ARM programs on an ARM emulator such as the QEMU (`www.qemu.org`) emulator, or *arm-elf-run* (available at `www.gnuarm.com`).