Lecture Two: Embedded ARM Microcontrollers

4.2) Addressing Modes and Operands

The **addressing mode** *refers to the various ways a processor can specify instructions to access data.* All instructions begin by fetching the machine instruction (op code and operand) pointed to by the PC. Some instructions operate completely within the processor and require no memory data fetches. For example, the **ADD R1, R2** instruction performs R1+R2 and stores the sum back into R1. If the data is found in the instruction itself, like **MOV R0, #1** , the instruction uses **immediate addressing** mode. A register that contains the address or the location of the data is called a **pointer** or **index** register. **Indexed addressing** mode uses a register pointer to access memory. The addressing mode that uses the PC as the pointer is called **PC-relative addressing** mode. It is used for branching, for calling functions, and accessing constant data stored in ROM.

The **MOV** instruction will move data within the processor without accessing memory.

The **LDR** instruction will read a 32-bit word from memory and place the data in a register.

With PC-relative addressing, the assembler automatically calculates the correct PC offset.

Register. Most instructions operate on the registers. In general, data flows towards the op code (right to left). In other words, the register closest to the op code gets the result of the operation. In each of these instructions, the result goes into R2.

```
MOV R2, #100    ; R2=100,      immediate addressing
LDR R2, [R1]    ; R2= value pointed to by R1
ADD R2, R0      ; R2= R2+R0
ADD R2, R0, R1  ; R2= R0+R1
```

Register list. The stack push and stack pop instructions can operate on one register or on a list of registers. SP is the same as R13, LR is the same as R14, and PC is the same as R15.

```
PUSH {LR}              ; save LR on stack
POP {LR}               ; remove from stack and place in LR
PUSH {R1-R3, LR}       ; save R1, R2, R3 and link register
POP {R1-R3, PC}        ; restore R1, R2, R3 and PC
```

A) Immediate addressing. With immediate addressing mode, the data itself is contained in the instruction. Once the instruction is fetched no additional memory access cycles are required to get the data. Notice the number 100 (0x64) is embedded in the machine code of the instruction shown

in Figure 2.10. Immediate addressing is only used to get, load, or read data. It will never be used with an instruction that stores to memory.

MOV R0, #100 ; R0=100, immediate addressing

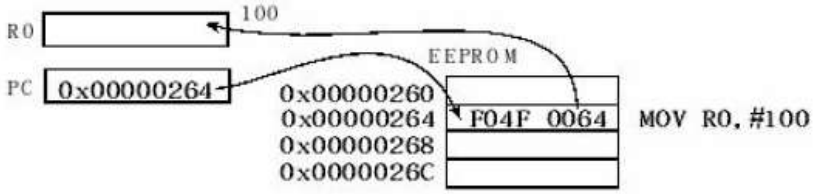

Figure 2.10. An example of immediate addressing mode, data is in the instruction.

B) Indexed addressing. With indexed addressing mode, the data is in memory and a register will contain a pointer to the data. Once the instruction is fetched, one or more additional memory access cycles are required to read or write the data. In these examples, R1 points to RAM.

LDR R0, [R1] ; R0= value pointed to by R1
LDR R0, [R1,#4] ; R0= word pointed to by R1+4
LDR R0, [R1, #4]! ; first R1=R1+4, then R0= word pointed to by R1
LDR R0, [R1], #4 ; R0= word pointed to by R1, then R1=R1+4
LDR R0, [R1, R2] ; R0= word pointed to by R1+R2
LDR R0, [R1, R2, LSL #2] ; R0= word pointed to by R1+4*R2

In Figure 2.11, R1 points to RAM, the instruction **LDR R0,[R1]** will read the 32-bit value pointed to by R1 and place it in R0. R1 could be pointing to any valid object in the memory map (i.e., RAM, ROM, or I/O), and R1 is not modified by this instruction.

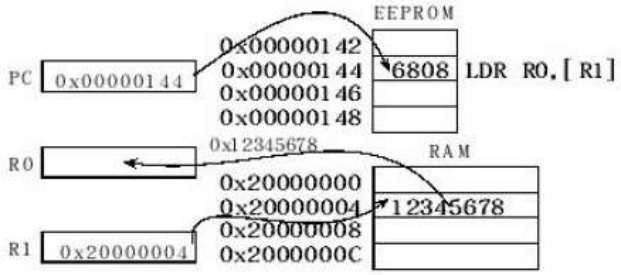

Figure 2.11. An example of indexed addressing mode, data is in memory.

In Figure 2.12, R1 points to RAM, the instruction **LDR R0,[R1,#4]** will read the 32-bit value pointed to by R1+4 and place it in R0. Even though the memory address is calculated as R1+4, the Register R1 itself is not modified by this instruction.

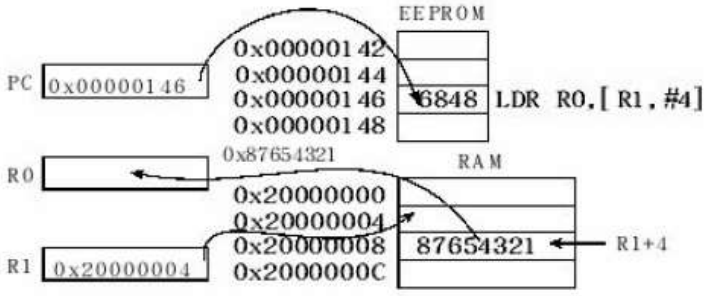

Figure 2.12. An example of indexed addressing mode with offset, data is in memory.

C) PC-relative addressing. PC-relative addressing is indexed addressing mode using the PC as the pointer. The PC always points to the instruction that will be fetched next, so changing the PC will cause the program to branch. A simple example of PC-relative addressing is the unconditional branch. In assembly language, we simply specify the label to which we wish to jump, and the assembler encodes the instruction with the appropriate PC-relative offset.

B Location ; jump to Location, using PC-relative addressing

The same addressing mode is used for a function call. Upon executing the **BL** instruction, the return address is saved in the link register (LR). In assembly language, we simply specify the label defining the start of the function, and the assembler creates the appropriate PC-relative offset.

BL Subroutine; call Subroutine, using PC-relative addressing

Typically, it takes two instructions to access data in RAM or I/O. The first instruction uses PC relative addressing to create a pointer to the object, and the second instruction accesses the memory using the pointer. We can use the **= something** operand for any symbol defined by our program. In this case **Count** is the label defining a 32-bit variable in RAM.

LDR R1, =Count ; R1 points to variable Count, using PC-relative
LDR R0, [R1] ; R0= value pointed to by R1

The operation caused by the above two **LDR** instructions is illustrated in Figure 2.13. Assume a 32-bit variable **Count** is located in the data space at RAM address 0x2000.0000.

First, **LDR R1, =Count** makes R1 equal to 0x2000.0000. I.e., R1 points to **Count**. The assembler places a constant 0x2000.0000 in code space and translates the **=Count** into the correct PC-relative access to the constant (e.g., **LDR R1, [PC, #28]**). In this case, the constant 0x2000.0000, the address of **Count**, will be located at PC+28. Second, the **LDR R0, [R1]** instruction will dereference this pointer, bringing the 32-bit contents at location 0x2000.0000 into R0. Since **Count** is located at 0x2000.0000, these two instructions will read the value of **Count** into R0.

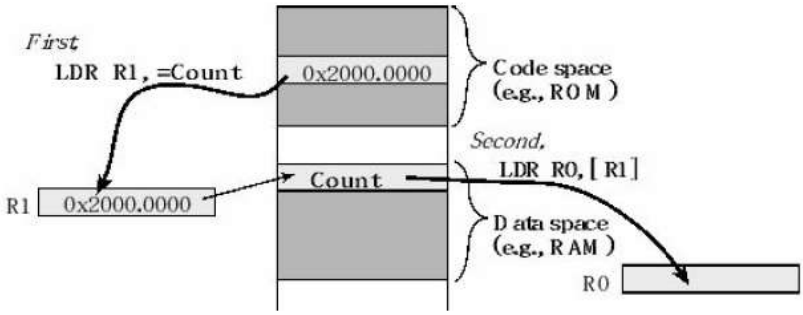

Figure 2.13. Indexed addressing using R1 as a register pointer to access memory. Data is moved into R0. Code space is where we place programs and data space is where we place variables.

Flexible second operand <op2>. Many instructions have a flexible second operand, shown as **<op2>** in the descriptions of the instruction. **<op2>** can be a constant or a register with optional shift. The flexible second operand can be a constant in the form **#constant**

ADD Rd, Rn, #constant ; Rd = Rn + constant

We can also specify a flexible second operand in the form **Rm {, shift}** . If **Rd** is missing, **Rn** is also the destination. For example:

ADD Rd, Rn, Rm {, shift} ; Rd = Rn+Rm
ADD Rn, Rm {, shift} ; Rn = Rn+Rm

where **Rm** is the register holding the data for the second operand, and **shift** is an optional shift to be applied to **Rm** . The optional **shift** can be one of these five formats:

ASR #n Arithmetic (signed) shift right **n** bits, $1 \leq \mathbf{n} \leq 32$.
LSL #n Logical (unsigned) shift left **n** bits, $1 \leq \mathbf{n} \leq 31$.
LSR #n Logical (unsigned) shift right **n** bits, $1 \leq \mathbf{n} \leq 32$.
ROR #n Rotate right **n** bits, $1 \leq \mathbf{n} \leq 31$.
RRX Rotate right one bit, with extend.
If we omit the shift, or specify **LSL #0** , the value of the flexible second operand is **Rm** . If we specify a shift, the shift is applied to the value in **Rm** , and the resulting 32-bit value is used by the instruction. However, the contents in the register **Rm** remain unchanged. For example,

```
ADD R0,R1,LSL #4      ; R0 = R0 + R1*16 (R1 unchanged)
ADD R0,R1,R2,ASR #4 ; signed R0 = R1 + R2/16 (R2 unchanged)
```

4.3) Memory Access Instructions

This section presents mechanisms to read from and write to memory. As illustrated in Figure 2.13, to access memory we first establish a pointer to the object, then use indexed addressing. Usually code space is in ROM, but it is possible to assign code space to RAM. Data space is where we place variables. There are four types of memory objects, and typically we use a specific register to access them.

Memory object type	**Register**	**Example operand**
Constants in code space	PC	**=Constant [PC, #28]**
Local variables on the stack	SP	**[SP, #0x04]**
Global variables in RAM	R0 – R12	**[R0]**
I/O ports	R0 – R12	**[R0]**

The **ADR** instruction uses PC-relative addressing and is a handy way to generate a pointer to a constant in code space or an address within the program. The general form for **ADR** is

ADR{cond} Rd, label

where **{cond}** is an optional condition (see Table 2.2), **Rd** is the destination register, and **label** is a label within the code space within the range of -4095 to +4095 from the address in the PC. In reality, the assembler will generate an **ADD** or **SUB** instruction to calculate the desired address using an offset to the PC. **DCD** is an assembler directive that defines a 32-bit constant. We use it to create constants in code space (ROM).

In the following example, after executing the **ADR** instruction, R5 points to **Pi** , and after executing the **LDR** instruction, R6 contains the data at **Pi** .

```
Access  ADR R5, Pi      ;R5 points to Pi
        LDR R6,[R5]          ;R6 = 314159
         ...
        BX LR
        Pi DCD 314159
```

We use the **LDR** instruction to load data from memory into a register. There is a special form of **LDR** which instructs the assembler to load a constant or address into a register. This is a "pseudo instruction" and the assembler will output suitable instructions to generate the specified value in the register. This form for **LDR** is

LDR{cond} Rd, =number

LDR{cond} Rd, =label

where **{cond}** is an optional condition (see Table 2.2), **Rd** is the destination register, and **label** is a label anywhere in memory. Figure 2.13 illustrates how to create a pointer to a variable in RAM. A similar approach can be used to access I/O ports. On the TM4C family, Port A exists at address 0x4000.43FC. After executing the first **LDR** instruction, R5 equals 0x4000.43FC, which is a pointer to Port A, and after executing the second **LDR** instruction, R6 contains the value at Port A at the time it was read.

```
Input LDR R5, =0x400043FC      ; R5=0x400043FC, R5 points to Port A
      LDR R6, [R5]                    ; Input from Port A into R6
      ; ...
      BX LR
```

We use the **LDR** instruction to load data from RAM to a register and the **STR** instruction to store data from a register to RAM. In most cases, it creates a copy of the data and places the copy at the new location. In other words, since the original data still exists in the previous location, there are now two copies of the information. The exception to this memory-access-creates-two-copies-rule is a stack pop. When we pop data from the stack, it no longer exists on the stack leaving us just one copy. For example in Figure 2.13, the instruction **LDR R0,[R1]** loads the contents of the variable **Count** into R0. At this point, there are two copies of the data, the original in RAM and the copy in R0. If we next add 1 to R0, the two copies have different values.

{type}	Data type	Meaning	
	32-bit word	0 to 4,294,967,295 or -2,147,483,648 to +2,147,483,647	
B	Unsigned 8-bit byte	0 to 255, bits on load	Zero pad to 32
SB	Signed 8-bit byte	-128 to +127, bits on load	Sign extend to 32
H	Unsigned 16-bit halfword	0 to 65535, bits on load	Zero pad to 32
SH	Signed 16-bit halfword	-32768 to +32767, bits on load	Sign extend to 32
D	64-bit data	registers	Uses two

Table 2.3. Optional modifier to specify data type when accessing memory.

Most of the addressing modes listed in the previous section can be used with load and store. The following lists the general form for some of the load and store instructions

LDR{type}{cond} Rd, [Rn] ; load memory at [Rn] to Rd

STR{type}{cond} Rt, [Rn] ; store Rt to memory at [Rn]

LDR{type}{cond} Rd, [Rn, #n] ; load memory at [Rn+n] to Rd

STR{type}{cond} Rt, [Rn, #n] ; store Rt to memory [Rn+n]

LDR{type}{cond} Rd, [Rn,Rm,LSL #n] ; load memory at [Rn+Rm<<n] to Rd

STR{type}{cond} Rt, [Rn,Rm,LSL #n] ; store Rt to memory [Rn+Rm<<n]

The move instructions get their data from the machine instruction or from within the processor and do not require additional memory access instructions.

MOV{S}{cond} Rd, <op2> ; set Rd equal to the value specified by op2

MOV{cond} Rd, #im16 ; set Rd equal to im16, im16 is 0 to 65535

MVN{S}{cond} Rd, <op2> ; set Rd equal to the -value specified by op2

4.4) Logical Operations

Software uses logical and shift operations to combine information, to extract information and to test information. A **unary operation** produces its result given a single input parameter. Examples of unary operations include negate, complement, increment, and decrement. In discrete digital logic, the **complement** operation is called a NOT gate as shown in Table 2.4.

A	~A
0	1
1	0

Table 2.4. Logical complement.

When designing digital logic we use gates, such as NOT AND OR, to convert individual input signals into individual output signals. However, when writing software using logic functions, we take two 32-bit numbers and perform 32 logic operations at the same time in a bit-wise fashion yielding one 32-bit result. Boolean Logic has two states: true and false. The false is 0, and the true state is any nonzero value. A **binary operation** produces a single result given two inputs. The **logical and** (&) operation yields a true result if both input parameters are true. The **logical or** (|) operation yields a true result if either input parameter is true. The **exclusive or** (^) operation yields a true result if exactly one input parameter is true. The logical operators are summarized in Table 2.5. The logical instructions on the ARM Cortex-M processor take two inputs, one from a register

and the other from the flexible second operand. These operations are performed in a bit-wise fashion on two 32-bit input parameters yielding one 32-bit output result. The result is stored into the destination register. For example, the calculation r=m&n means each bit is calculated separately, $r_{31}=m_{31}\&n_{31}$, $r_{30}=m_{30}\&n_{30}$, …, $r_0=m_0\&n_0$.

In C, when we write **r=m&n; r=m|n; r=m^n;** the logical operation occurs in a bit-wise fashion as described by Table 2.5. However, in C we define the Boolean functions as **r=m&&n; r=m||n;** For Booleans, the operation occurs in a word-wise fashion. For example, **r=m&&n;** means **r** will become zero if either **m** is zero or **n** is zero. Conversely, **r** will become 1 if both **m** is nonzero and **n** is nonzero.

A **Rn**	B **Operand2**	A&B **AND**	A|B **ORR**	A^B **EOR**	A&(~B) **BIC**	A|(~B) **ORN**
0	0	0	0	0	0	1
0	1	0	1	1	0	0
1	0	0	1	1	1	1
1	1	1	1	0	0	1

Table 2.5. Logical operations performed by the Cortex -M processor.

All instructions place the result into the destination register **Rd**. If **Rd** is omitted, the result is placed into **Rn**, which is the register holding the first operand. If the optional **S** suffix is specified, the N and Z condition code bits are updated on the result of the operation. In the comments next to the instructions below, we use **op2** to represent the 32-bit value generated by the flexible second operand, **<op2>**. Some flexible second operands may affect the C bit. These logical instructions will leave the V bit unchanged.

AND{S}{cond} {Rd,} Rn, <op2> ;Rd=Rn&op2

ORR{S}{cond} {Rd,} Rn, <op2> ;Rd=Rn|op2

EOR{S}{cond} {Rd,} Rn, <op2> ;Rd=Rn^op2

BIC{S}{cond} {Rd,} Rn, <op2> ;Rd=Rn&(~op2)

ORN{S}{cond} {Rd,} Rn, <op2> ;Rd=Rn|(~op2)

For example, assume R1 is 0x12345678 and R2 is 0x87654321. The **ORR R0,R1,R2** will perform this operation, placing the 0x97755779 result in R0.

R1 0001 0010 0011 0100 0101 0110 0111 1000

R2 1000 0111 0110 0101 0100 0011 0010 0001

ORR 1001 0111 0111 0101 0101 0111 0111 1001

Example 2.1: Write code to set bit 0 in a 32-bit variable called **N**.

Solution: First, we perform a 32-bit read, bringing **N** into Register R0. Second we perform a logical OR setting bit 0, and lastly we store the result back into **N**.

```
LDR R1, =N      ; R1 = &N (R1 points to N)
LDR R0, [R1]    ; R0 = N
ORR R0, R0, #1 ; R0 = N|1
STR R0, [R1]    ; N = N|1
```

```
// C implementation
N = N | 0x00000001;
```

Program 2.1. Example code showing a logical OR.

Observation: We use the logical OR to make bits become one, and we use the logical AND to make bits become zero.

4.5) Shift Operations

Like programming in C, the assembly shift instructions take two input parameters and yield one output result. In C, the left shift operator is << and the right shift operator is >>. E.g., to left shift the value in M by N bits and store the result in R we execute: R = M<<N. Similarly, to right shift the value in M by N bits and store the result in R we execute: R = M>>N.

The **logical shift right** (LSR) is similar to an unsigned divide by 2^n, where n is the number of bits shifted as shown in Figure 2.14. A zero is shifted into the most significant position, and the carry flag will hold the last bit shifted out. The right shift operations do not round. For example, a right shift by 3 bits is similar to divide by 8. However, 15 right-shifted three times (15>>3) is 1.

The **arithmetic shift right** (ASR) is similar to a signed divide by 2^n. Notice that the sign bit is preserved, and the carry flag will hold the last bit shifted out. This right shift operation also does not round. Again, a right shift by 3 bits is similar to divide by 8. However, -9 right-shifted three times (-9>>3) is -2.

The **logical shift left** (LSL) operation works for both unsigned and signed multiply by 2^n. A zero is shifted into the least significant position, and the carry bit will contain the last bit that was shifted out.

The two **rotate** operations can be used to create multiple-word shift functions. There is no rotate left instruction, because a rotate left 10 bits is the same as rotate right 22 bits. All shift instructions place the result into the destination register **Rd** . **Rm** is the register holding the value to be shifted. The number of bits to shift is either in register **Rs** , or specified as a constant **n** . If the optional **S** suffix is specified, the N and Z condition code bits are updated on the result of the operation. The

C bit is the carry out after the shift as shown in Figure 2.14. These shift instructions will leave the V bit unchanged.

Observation: Use logic shift for unsigned numbers and arithmetic shifts for signed numbers.

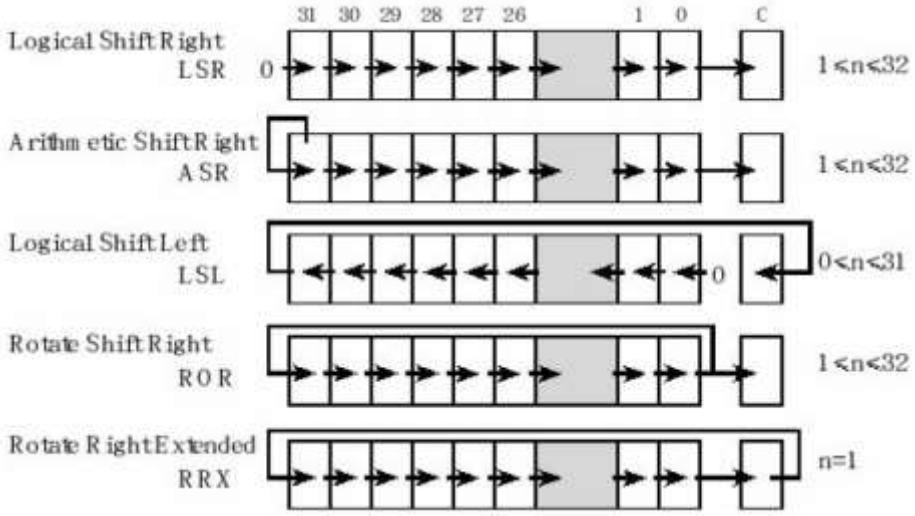

Figure 2.14. Shift operations.

LSR{S}{cond} Rd, Rm, Rs ; logical shift right Rd=Rm>>Rs (unsigned)
LSR{S}{cond} Rd, Rm, #n ; logical shift right Rd=Rm>>n (unsigned)
ASR{S}{cond} Rd, Rm, Rs ; arithmetic shift right Rd=Rm>>Rs (signed)
ASR{S}{cond} Rd, Rm, #n ; arithmetic shift right Rd=Rm>>n (signed)
LSL{S}{cond} Rd, Rm, Rs ; shift left Rd=Rm<<Rs (signed, unsigned)
LSL{S}{cond} Rd, Rm, #n ; shift left Rd=Rm<<n (signed, unsigned)
ROR{S}{cond} Rd, Rm, Rs ; rotate right
ROR{S}{cond} Rd, Rm, #n ; rotate right
RXX{S}{cond} Rd, Rm ; rotate right 1 bit with extension

Example 2.2: Write code that reads from variable **N**, shifts right twice, and stores the result in variable **M**. Both variables are 32-bit unsigned.

Solution: First, we perform a 32-bit read, bringing **N** into Register R1. Second we divide by 4 using a shift right operation, and lastly we store the result into **M**. Since the value gets smaller, no overflow can occur. If the variables were signed, then the **LSR** instruction should be replaced with an **ASR** instruction. In C, the shift right operator is >>

```
LDR R3, =N      ; R3 = &N (R3 points to      // C implementation
N)                                            M = N>>2;
  LDR R1, [R3]    ; R1 = N
  LSR R0, R1, #2  ; R0 = N>>2
  LDR R2, =M      ; R2 = &M (R2 points
to M)
  STR R0, [R2]    ; M = N>>2
```

Program 3.2. Example code showing a right shift.

Example 3.3: Assume we have three 8-bit variables named **High** , **Low** , and **Result** . **High** and **Low** have 4 bits of data; each is a number from 0 to 15. Take these two 4-bit nibbles and combine them into one 8-bit value, storing the combination in **Result** .

Solution: The solution uses the **shift** operation to move the bits into position, then it uses the logical **OR** operation to combine the two parts into one number. We will assume both **High** and **Low** are bounded within the range of 0 to 15. The expression **High<<4** will perform four logical shift lefts. Registers R2, R3, and R4 point to (contain the address of) variables.

```
LDR R2, =High  ; R2 = &High          // C implementation
LDR R3, =Low   ; R3 = &Low           Result = (High<<4)|Low;
LDR R4, =Result ; R4 = &Result
LDRB R1, [R2]   ; R1 = High
LSL R0, R1, #4  ; R0 = R1<<4 =
High<<4
LDRB R1, [R3]   ; R1 = Low
ORR R0, R0, R1  ; R0 =
(High<<4)|Low
STRB R0, [R4]   ; Result =
(High<<4)|Low
```

Program 2.3. Example code showing a left shift.

To illustrate how the above program works, let 0 0 0 0 h3 h2 h1 h0be the value of **High** , and let 0 0 0 0 l3 l2 l1 l0be the value of **Low** . The **LDRB R1,[R2]** instruction brings the 8-bit **High** into Register R1.

The **LSL R0,R1,#4** instruction moves the **High** into bit positions 4-7 of Register R0. The **LDRB R1, [R3]** instruction brings the 8-bit **Low** into Register R1. Finally, the **ORR R0,R0,R1** instruction combines **High** and **Low** , and the **STRB R0,[R4]** instruction stores the combination into **Result** .

0 0 0 0 h3 h2 h1 h0 value of **High**

h3 h2 h1 h0 0 0 0 0 after four **LSL** s

0 0 0 0 l3 l2 l1 l0 value of **Low**

h3 h2 h1 h0 l3 l2 l1 l0 result of the **ORR** instruction