Addressing Modes and Operands

A fundamental issue in program development is the differentiation between data and address. When we put the number 1000 into Register R0, whether this is data or address depends on how the 1000 is used. To run efficiently, we try to keep frequently accessed information in registers. However, we need to access memory to fetch parameters or save results. The **addressing mode** is the format the instruction uses to specify the memory location to read or write data. The addressing mode is associated more specifically with the operands, and a single instruction could exercise multiple addressing modes for each of the operands. When the import is obvious though, we will use the expression “the addressing mode of the instruction”, rather than “the addressing mode of an operand in an instruction". All instructions begin by fetching the machine instruction (op code and operand) pointed to by the PC. When extended with Thumb-2 technology, some machine instructions are 16 bits wide, while others are 32 bits. 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 addressing mode is called PC relative because the machine code contains the address difference between where the program is now and the address to which the program will access. 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,R2,LR}   ; save registers and return address
POP   {R1,R2,PC}   ; restore registers and return
```

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 2.14. 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
```

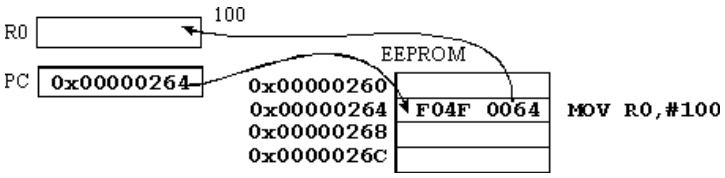

Figure 2.14. An example of immediate addressing mode, data is in the instruction.

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. In this class, we will focus on just the first two forms of indexed addressing.

Help

```
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 2.15, 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.

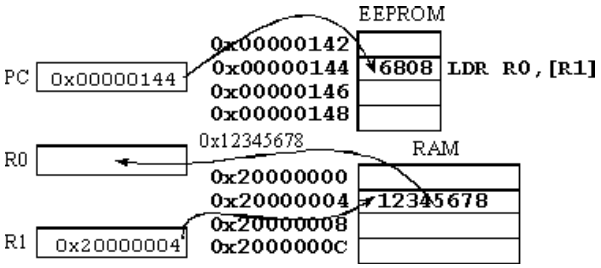

Figure 2.15. An example of indexed addressing mode, data is in memory.

In Figure 2.16, 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.

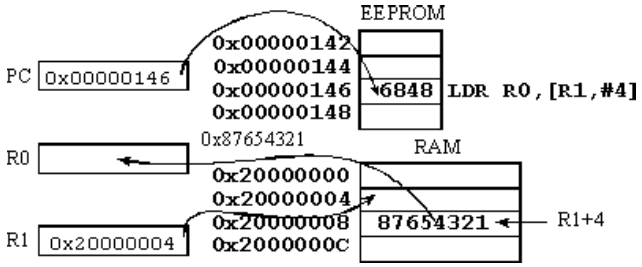

Figure 2.16. An example of indexed addressing mode with offset, data is in memory.

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.

Help

```
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 2.17. 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 the variable into R0.

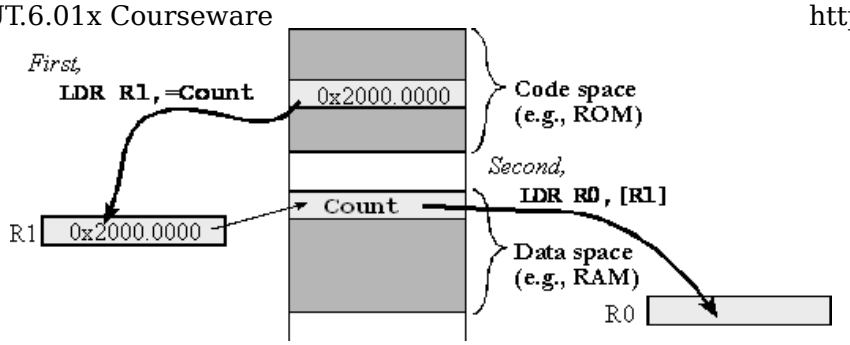

Figure 2.17. Indexed addressing using R1 as a register pointer to access memory. Data is moved into R0. Code space is where we place programs and data space is where we place variables.

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
```

where **constant** is calculated as one of these four, **X** and **Y** are hexadecimal digits:

- Constant produced by shifting an unsigned 8-bit value left by any number of bits
- Constant of the form **0x00XY00XY**
- Constant of the form **0xXY00XY00**
- Constant of the form **0xXYXYXYXY**

Help

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, $1 \le n \le 32$.
LSL #n Logical (unsigned) shift left **n** bits, $1 \le n \le 31$.
LSR #n Logical (unsigned) shift right **n** bits, $1 \le n \le 32$.
ROR #n Rotate right **n** bits, $1 \le n \le 31$.
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)
```

An **aligned access** is an operation where a word-aligned address is used for a word, dual word, or multiple word access, or where a halfword-aligned address is used for a halfword access. Byte accesses are always aligned. The address of an aligned word access will have its bottom two bits equal to zero. An **unaligned** word access means we are accessing a 32-bit object (4 bytes) but the address is not evenly divisible by 4. The address of an aligned halfword access will have its bottom bit equal to zero. An unaligned halfword access means we are accessing a 16-bit object (2 bytes) but the address is not evenly divisible by 2. The Cortex-M processor supports unaligned access only for the following instructions:

LDR	Load 32-bit word
LDRH	Load 16-bit unsigned halfword
LDRSH	Load 16-bit signed halfword (sign extend bit 15 to bits 31-16)
STR	Store 32-bit word
STRH	Store 16-bit halfword

Help

Transfers of one byte are allowed for the following instructions:

LDRB	Load 8-bit unsigned byte
LDRSB	Load 8-bit signed byte (sign extend bit 7 to bits 31-8)
STRB	Store 8-bit byte

When loading a 32-bit register with an 8- or 16-bit value, it is important to use the proper load, depending on whether the number being loaded is signed or unsigned. This determines what is loaded into the most significant bits of the register to ensure that the number keeps the same value when it is promoted to 32 bits. When loading an 8-bit unsigned number, the top 24 bits of the register will become zero. When loading an 8-bit signed number, the top 24 bits of the register will match bit 7 of the memory data (signed extend). Note that there is no such thing as a signed or unsigned store. For example, there is no **STRSH**; there is only **STRH**. This is because 8, 16, or all 32 bits of the register are stored to an 8-, 16-, or 32-bit location, respectively. No promotion occurs. This means that the value stored to memory can be different from the value located in the register if there is overflow. When using **STRB** to store an 8-bit number, be sure that the number in the register is 8 bits or less.

All other read and write memory operations generate a usage fault exception if they perform an unaligned access, and therefore their accesses must be address aligned. Also, unaligned accesses are usually slower than aligned accesses, and some areas of memory do not support unaligned accesses. But unaligned accesses may allow programs to use memory more efficiently at the cost of performance. The tradeoff between speed and size is a common motif.

Observation: We add a dot in the middle of 32-bit hexadecimal numbers (e.g., 0x2000.0000). This dot helps the reader

visualize the number. However, this dot should not be used when writing actual software.

CHECKPOINT 2.33

What is the addressing mode used for?

Hide Answer

The addressing mode defines the format for the effective address for that instruction. In other words, it defines how the instruction will access the data it needs.

CHECKPOINT 2.34

Assume R3 equals 0x2000.0000 at the time **`LDR R2,[R3,#8]`** is executed. What address will be accessed? If R3 is changed, to what value will R3 become?

Hide Answer

0x2000.0008, R3 is not changed.

Help

CHECKPOINT 2.35

Assume R3 equals 0x2000.0000 at the time **`LDR R2,[R3],#8`** is executed. What address will be accessed? If R3 is changed, to what value will R3 become?

Hide Answer

0x2000.0000, and 8 is added to R3, it becomes 0x2000.0008.