**Branch Instructions:**

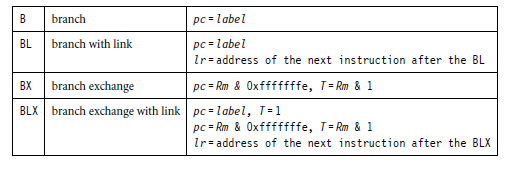
A branch instruction changes the flow of execution or is used to call a routine. This type of instruction allows programs to have subroutines, *if-then-else* structures, and loops. The change of execution flow forces the program counter *pc* to point to a new address. The ARMv5E instruction set includes four different branch instructions.

Syntax: B{<cond>} label

BL{<cond>} label

BX{<cond>} Rm

BLX{<cond>} label | Rm



The address *label* is stored in the instruction as a signed *pc*-relative offset and must be within approximately 32 MB of the branch instruction. *T* refers to the Thumb bit in the *cpsr*. When instructions set *T*, the ARM switches to Thumb state.

**Example 1:**

This example shows a forward and backward branch. Because these loops are address specific, we do not include the pre- and post-conditions. The forward branch skips three instructions. The backward branch creates an infinite loop.

B forward

ADD r1, r2, #4

ADD r0, r6, #2

ADD r3, r7, #4

forward

SUB r1, r2, #4

**………………………………..**

backward

ADD r1, r2, #4

SUB r1, r2, #4

ADD r4, r6, r7

B backward

Branches are used to change execution flow. Most assemblers hide the details of a branch instruction encoding by using labels. In this example, *forward* and *backward* are the labels. The branch labels are placed at the beginning of the line and are used to mark an address that can be used later by the assembler to calculate the branch offset.

**Example 2:**

The branch with link, or BL, instruction is similar to the B instruction but overwrites the link register *lr* with a return address. It performs a subroutine call. This example shows a simple fragment of code that branches to a subroutine using the BL instruction. To return from a subroutine, you copy the link register to the *pc*.

BL subroutine ; branch to subroutine

CMP r1, #5 ; compare r1 with 5

MOVEQ r1, #0 ; if (r1==5) then r1 = 0

:

subroutine

<subroutine code>

MOV pc, lr ; return by moving pc = lr

The branch exchange (BX) and branch exchange with link (BLX) are the third type of branch instruction. The BX instruction uses an absolute address stored in register *Rm*. It is primarily used to branch to and from Thumb code, as shown in Chapter 4. The *T* bit in the *cpsr* is updated by the least significant bit of the branch register. Similarly the BLX instruction updates the *T* bit of the *cpsr* with the least significant bit and additionally sets the link register with the return address.

**Load-Store Instructions:**

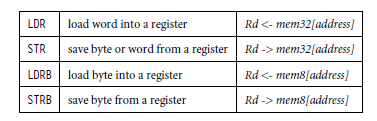
Load-store instructions transfer data between memory and processor registers. There are three types of load-store instructions: single-register transfer, multiple-register transfer, and swap.

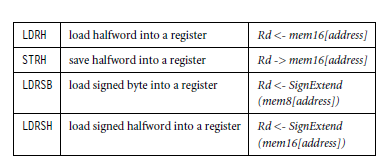
Single-Register Transfer:

Syntax: <LDR|STR>{<cond>}{B} Rd,addressing1

LDR{<cond>}SB|H|SH Rd, addressing2

STR{<cond>}H Rd, addressing2





**Example 1:**

LDR and STR instructions can load and store data on a boundary alignment that is the same as the datatype size being loaded or stored. For example, LDR can only load 32-bit words on a memory address that is a multiple of four bytes—0, 4, 8, and so on. This example shows a load from a memory address contained in register *r1*, followed by a store back to the same address in memory.

; load register r0 with the contents of

; the memory address pointed to by register

; r1.

;

LDR r0, [r1] ; = LDR r0, [r1, #0]

;

; store the contents of register r0 to

; the memory address pointed to by

; register r1.

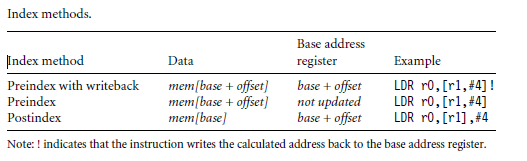
;

STR r0, [r1] ; = STR r0, [r1, #0]

The first instruction loads a word from the address stored in register *r1* and places it into register *r0*. The second instruction goes the other way by storing the contents of register *r0* to the address contained in register *r1*. The offset from register *r1* is zero. Register *r1* is called the *base address register*.

Single-Register Load-Store Addressing Modes:

The ARM instruction set provides different modes for addressing memory. These modes incorporate one of the indexing methods: preindex with writeback, preindex, and postindex (see Table 1).



**Example 2:**

Preindex with writeback calculates an address from a base register plus address offset and then updates that address base register with the new address. In contrast, the preindex offset is the same as the preindex with writeback but does not update the address base register. Postindex only updates the address base register after the address is used. The preindex mode is useful for accessing an element in a data structure. The postindex and preindex with writeback modes are useful for traversing an array.

**PRE**

r0 = 0x00000000

r1 = 0x00090000

mem32[0x00009000] = 0x01010101

mem32[0x00009004] = 0x02020202

**Preindexing with writeback:**

LDR r0, [r1, #4]!

**POST(1)**

r0 = 0x02020202

r1 = 0x00009004

**Preindexing:**

LDR r0, [r1, #4]

**POST(2)**

r0 = 0x02020202

r1 = 0x00009000

**Postindexing:**

LDR r0, [r1], #4

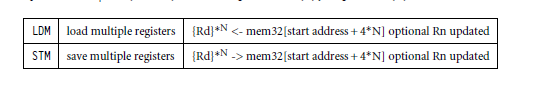
**POST(3)**

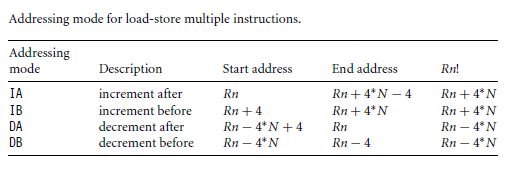
r0 = 0x01010101

r1 = 0x00009004

Multiple-Register Transfer:

Load-store multiple instructions can transfer multiple registers between memory and the processor in a single instruction. The transfer occurs from a base address register *Rn*pointing into memory. Multiple-register transfer instructions are more efficient from single-register transfers for moving blocks of data around memory and saving and restoring context and stacks.





**Example 3:**

In this example, register *r0* is the base register *Rn* and is followed by !, indicating that the register is updated after the instruction is executed. You will notice within the load multiple instruction that the registers are not individually listed. Instead the “-” character is used to identify a range of registers. In this case the range is from register *r1* to *r3* inclusive.

Each register can also be listed, using a comma to separate each register within “{” and “}” brackets.

**PRE**

mem32[0x80018] = 0x03

mem32[0x80014] = 0x02

mem32[0x80010] = 0x01

r0 = 0x00080010

r1 = 0x00000000

r2 = 0x00000000

r3 = 0x00000000

LDMIA r0!, {r1-r3}

**POST**

r0 = 0x0008001c

r1 = 0x00000001

r2 = 0x00000002

r3 = 0x00000003