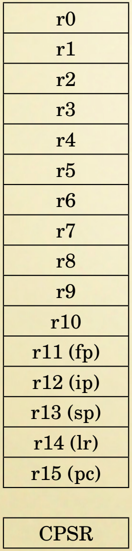
Modern Assembly Language Programming with the ARM processor

Chapter 3: Load/Store and Branch Instructions

1. Introduction
2. Load/Store Instructions
3. Branch Instructions

ARM User Program Registers



- Thirteen general-purpose registers (`r0–r12`)
- The stack pointer (`r13` or `sp`)
- The link register (`r14` or `lr`)
- The program counter (`r15` or `pc`)
- Current Program Status Register (`CPSR`)

Hardware-Related Register Rules

- All instructions can access `r0-r14` directly.
- Most instructions also allow use of the program counter (`r15`).
- Specific instructions to allow access to `CPSR`.
- `r14`, `r15`, and `CPSR` are "hardware special".

CPSR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
N	Z	C	V	Q			J					GE[3:0]										E	A	I	F	T	M[4:0]				

Negative:	This bit is set to one if the signed result of an operation is negative, and set to zero if the result is positive or zero.
Zero:	This bit is set to one if the result of an operation is zero, and set to zero if the result is non-zero.
Carry:	This bit is set to one if an add operation results in a carry out of the most significant bit, or if a subtract operation results in a borrow. For shift operations, this flag is set to the last bit shifted out by the shifter.
oVerflow:	For addition and subtraction, this flag is set if a signed overflow occurred.

Conditional Execution

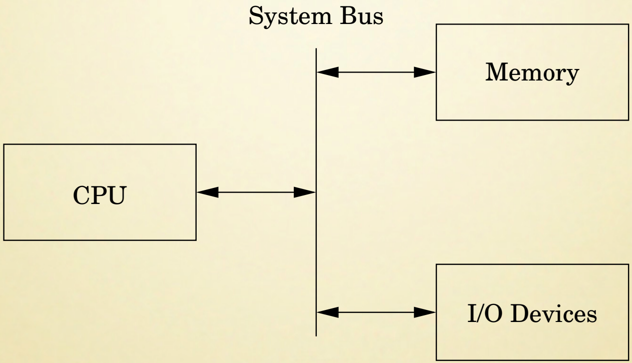
```
    op{<cond>}  operands
```

<cond>	English meaning
al	always (this is the default <cond>
eq	Z set (=)
ne	Z clear (≠)
ge	N set and V set, or N clear and V clear (≥)
lt	N set and V clear, or N clear and V set (<)
gt	Z clear, and either N set and V set, or N clear and V set (>)
le	Z set, or N set and V clear,or N clear and V set (≤)
hi	C set and Z clear (unsigned >)
ls	C clear or Z (unsigned ≤)
hs	C set (unsigned ≥)
cs	Alternate name for HS
lo	C clear (unsigned <)
cc	Alternate name for LO
mi	N set (result < 0)
pl	N clear (result ≥ 0)
vs	V set (overflow)
vc	V clear (no overflow)

Instruction Categories

- Load/Store Instructions
- Branch Instructions
    - Branch with Link (subroutine call)
    - Conditional Branches
- Data processing Instructions
    - Arithmetic Operations
    - Logical Operations
    - Comparison Operations
    - Data Movement Operations
    - Multiplication Operations
- Special Instructions
- Pseudo-Instructions

Simplified Computer



Pointers and Addresses

Data must be copied to a register before it can be used in any calculation, but there are not many registers.

In assembly, *almost all* data is accessed using its address in memory.

1. Every memory location has an address.
2. A pointer is a variable that holds an address.
3. A pointer can be stored in a register (short term) or in memory (long term).
4. Before it can be used to access the data it points to, a pointer variable must be loaded into a register.

The address of a statically allocated variable, `x`, can be loading using the following pseudo-instruction:

```
ldr r4, =x
```

This creates a temporary pointer variable in register `r4`, which can then be used to load data from variable `x`.

Addressing Modes

Most of the Load/Store instructions use an <address> which is one of the ten options listed below.

Syntax	Name
[Rn]	Register immediate
[Rn, #± <offset_12>]	Immediate offset
[Rn, ±Rm]	Register offset
[Rn, ±Rm, <shift> #<shift_imm>]	Scaled register offset
[Rn, #±<offset_12>]!	Immediate pre-indexed
[Rn, ±Rm]!	Register pre-indexed
[Rn, ±Rm, <shift> #<shift_imm>]!	Scaled register pre-indexed
[Rn], #±<offset_12>	Immediate post-indexed
[Rn], ±Rm	Register post-indexed
[Rn], ±Rm, <shift> #<shift_imm>	Scaled register post-indexed

<shift> can be any of the shift or rotate operations that will be covered later.

[Rn] is just shorthand notation for [Rn, #0]

Load/Store Operations

- Operations:

LDR / STR	Load/Store 32 bits
LDRH / STRH	Load/Store 16 bits unsigned
LDRB / STRB	Load/Store 8 bits unsigned
LDRSH	Load 16 bits signed
LDRSB	Load 8 bits signed

- Syntax:

```
    <Opcode>{<cond>}{<size>}  Rd, <address>
```

- Examples

```
ldrsh   r5, [r2]        @ Load r5 with signed
                        @ half-word at the address in r2
strb    r1, [r9, #4]    @ Store the byte in r1 at
                        @ the address (r9 + 4)
ldr     r7, [r3, r2]!   @ Load r5 with word at the
                        @ address (r3 + r2), then
                        @ store the address in r3
```

Load/Store Examples

```
ldrh    r9, [r2, #2]!  @ Load r9 with halfword at the
                       @ address (r2 + 2), then store
                       @ the address in r2
ldrsh   r5, [r2]       @ Load r5 with signed
                       @ half-word at the address in r2
strb    r1, [r9, #4]   @ Store the byte in r1 at
                       @ the address (r9 + 4)
ldr     r7, [r3, r2]!  @ Load r7 with word at the
                       @ address (r3 + r2), then
                       @ store the address in r3
ldrh    r9, [r2, #2]!  @ Load r9 with halfword at the
                       @ address (r2 + 2), then store
                       @ the address in r2
ldr     r7, [r3], #4   @ Load r7 with word at the
                       @ address in r3 then increment
                       @ r3 by 4
```

Load/Store Multiple Registers

These instructions are used to store registers on the stack, and for copying blocks of data. There are four variants for the LDM and STM instructions, and each variant has two equivalent names.

- Operations:
  LDM / STM   Load/Store Multiple Registers
- Syntax:
  LDM|STM{<variant>}   Rd{!}, {<list>}ˆ
- The trailing ˆ can only be used by operating system code.
- <variant> is chosen from the following table:

Block Copy		Stack	
IA	Increment After	EA	Empty Ascending
IB	Increment Before	FA	Full Ascending
DA	Decrement After	ED	Empty Descending
DB	Decrement Before	FD	Full Descending

Load/Store Multiple Registers (Continued)

The C compiler always uses the `stmfd` and `ldmfd` versions for the stack.

- Examples

```
stmfd sp!,{r4-r7,fp,lr}@ store r4, r5, r6, r7, r11, and
                       @ r14 on the stack, and store the
                       @ new stack pointer in sp
ldmfd sp!,{r4-r7,fp,lr}@ load r4, r5, r6, r7, r11, and
                       @ r14 from the stack, and store
                       @ the new stack pointer in sp
stmib r9!,{r0-r7}      @ Store 8 registers at the
                       @ location pointed to by r9, and
                       @ increment r9 BEFORE each store.
                       @ After executing, r9 will point
                       @ to the last item stored.
ldmia r4,{r0,r2,r3}    @ Load r0, r2, and r3 at the
                       @ location pointed to by r4 and
                       @ increment the address AFTER
                       @ each store. After executing, r4
                       @ will contain its original value.
```

Block Copy Example

```
        .data
source: .word  12
        .word  23
        .word  43
        .word  33
        .word  12
        .word  23
        .word  6
        .word  13
dest:   .skip  32
```

⋮

```
        stmfd   sp!{r0-r9} @ push r0...r9 to the stack
        ldr     r8,=source @ load address of source
        ldr     r9,=dest   @ load address of destination
        ldmia   r8,{r0-r7} @ load eight words from source
        stmia   r9,{r0-r7} @ store them in destination
        ldmfd   sp!{r0-r9} @ restore contents of r0...r9
```

Atomic Load-Store

Multiprogramming and threading require the ability to set and test values *atomically*. These instructions are used by the Operating System and/or threading libraries to guarantee *mutual exclusion*.

- Operations:

SWP	Load a word and store a word in one atomic operation.
SWPB	Load a byte and store a byte in one atomic operation.

- Syntax:
  SWP{<cond>}{B} Rd, Rm, [Rn]
- Example

```
swpeqb  r1, r4, [r3]  @ if (EQ) then load r1 with byte
                      @ at address in r3 and store byte
                      @ in r4 at address in r3
```

Note: SWP and SWPB are deprecated in favor of LDREX and STREX.

Mutual Exclusion - New Method

Exclusive load (`ldrex`) reads data from memory, tagging the memory address at the same time. Exclusive store (`strex`) stores data to memory, but only if the tag is still valid. Every memory access to the same address between `ldrex` and `strex` will invalidate the tag. This provides mutual exclusion on multiprocessor systems.

- Operations:

LDREX	Load register from memory and tag the memory address
STREX	Store register in memory if tag is valid, and report success

- Syntax:
  LDREX|STREX{<cond>}{{S}<size>}   Rd, <address>
- Example

```
      ldr     r12, =sem      @ preload semaphore address
      ldr     r1, =LOCKED    @ preload "locked" value
spin_lock:
      ldrex   r0, [r12]      @ load semaphore value
      cmp     r0, r1         @ if semaphore is not locked
      strexne r0, r1, [r12]  @    try to claim
      cmpne   r0, #1         @    and check success
      beq     spin_lock      @ retry if claiming failed or
                             @ it was already locked.
```

Branch Instructions

- Operations:

B	load pc with new address (branch)
BL	Save pc in lr, then load pc with new address (branch and link)

- Syntax:
  B{L}{<cond>}   <target_address>
- Example

```
     mov  r4, #10    @ load 10 into r4
loop_a:
     mov  r0, #1     @ fd -> stdout
     ldr  r1, =msg   @ buf -> msg
     ldr  r2, =len   @ count -> len(msg)
     mov  r7, #4     @ write is syscall #4
     swi  #0         @ invoke syscall
     subs r4, #1     @ decrement loop counter
     bne  loop_a     @ repeat until loop counter is zero
```