ARM Load/Store Instructions

- The ARM is a Load/Store Architecture:
  - Only load and store instructions can access memory
  - Does not support memory to memory data processing operations.
  - Must move data values into registers before using them.

ARM Load/Store Instructions

- ARM has three sets of instructions which interact with main memory. These are:
  - Single register data transfer (LDR/STR)
  - Block data transfer (LDM/STM)
  - Single Data Swap (SWP)

ARM Single Register Load/Store Instructions

- The basic load and store instructions are:

LDR	STR	Word
LDRB	STRB	Byte
LDRH	STRH	Halfword
LDRSB		Signed byte load
LDRSH		Signed halfword load

ARM Single Register Load/Store Instructions

- Memory system must support all access sizes
- Syntax:
  - LDR{<cond>}{<size>} Rd, <address>
  - STR{<cond>}{<size>} Rd, <address>

  e.g.

  ```
  LDR R0, [R1]

  STR R0, [R1]

  LDREQB  R0, [R1]
  ```

Data Transfer: Memory to Register (load)

- To transfer a word of data, we need to specify two things:
  - Register: r0 - r15
  - Memory address: more difficult
    - Think of memory as a single one-dimensional array, so we can address it simply by supplying a pointer to a memory address.
    - There are times when we will want to offset from this pointer.

ARM Addressing Modes

There are basically two types of addressing modes available in ARM

- Pre-indexed addressing: the address generated is used immediately
- Post-indexed addressing: the address generated later replaces the base register

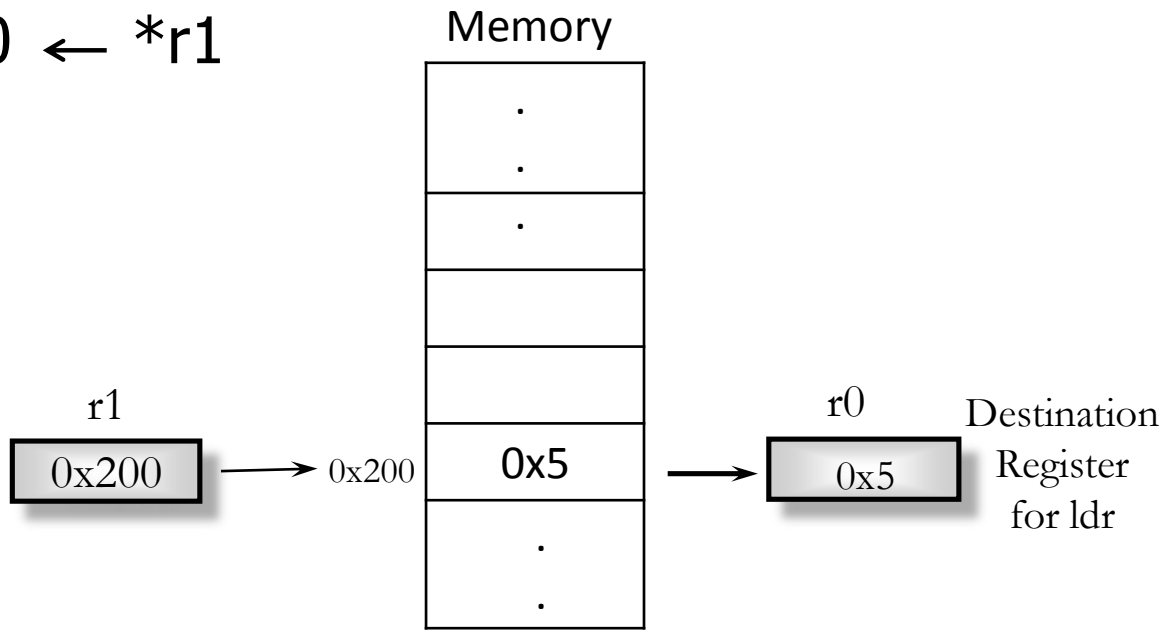
ARM Addressing Modes

[Rn] *Register*

Address accessed is value found in Rn.

Example:

```
ldr r0, [r1]      @  r0 ← *r1
```



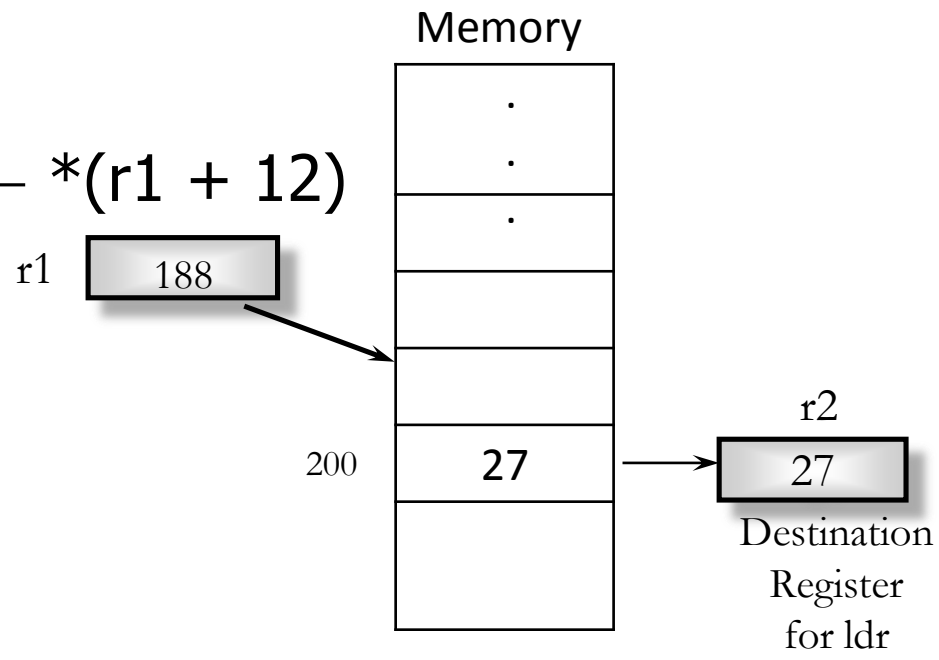
ARM Addressing Modes (Pre-Indexing)

[Rn, #±imm] *Immediate offset*

Address accessed is *imm* more/less than the address found in R*n*. R*n* does not change.

Example:

ldr r2, [r1, #12] @ r2 ← *(r1 + 12)



ARM Addressing Modes (Pre-Indexing)

[Rn, ±Rm]

Register offset

Address accessed is the value in R*n* ± the value in R*m*. R*n* and R*m* do not change values.

Example:

```
ldr r2, [r0, r1]     @  r2 ← *(r0 + r1)
```

ARM Addressing Modes (Pre-Indexing)

[Rn, ±Rm, shift] *Scaled register offset*

Address accessed is the value in Rn ± the value in Rm shifted as specified. Rn and Rm do not change values.

Example:

```
ldr r0, [r1, r2, lsl #2]        @  r0 ← *(r1 + r2*4)
```

ARM Addressing Modes (Pre-Indexing w\ update

[R*n*, #±*imm*]! *Immediate pre-indexed* w\update

Address accessed is as with *immediate offset* mode, but R*n*'s value updates to become the address accessed.

Example:

```
ldr r2, [r1, #12]!    @  r1 ← r1 + 12   then r2 ← *r1
```

ARM Addressing Modes (Pre-Indexing w\ update

[R*n*, ±R*m*]!

Register pre-indexed w\update

Address accessed is as with *register offset* mode, but R*n*'s value updates to become the address accessed.

Example:

```
ldr r2, [r0, r1]!      @  r0 ← r0 + r1   then r2 ← *r0
```

ARM Addressing Modes (Pre-Indexing w\ upadate

[R*n*, ±R*m*, *shift*]! *Scaled register pre-indexed* w\update

Address accessed is as with *scaled register offset* mode, but R*n*'s value updates to become the address accessed.

Example:

```
ldr r2, [r0, r1, lsl #2]!   @  r0 ← r0 + r1*4   then r2 ← *r0
```

ARM Addressing Modes (Post-Indexing)

[R*n*], #±*imm* *Immediate post-indexed*

Address accessed is value found in R*n*, and then R*n*'s value is increased/decreased by *imm*.

Example:

```
str r2, [r1], +4        @  *r1 ← r2   then   r1 ← r1 + 4
```

ARM Addressing Modes (Post-Indexing)

[R*n*], ±R*m* *Register post-indexed*

Address accessed is value found in R*n*, and then R*n*'s value is increased/decreased by R*m*.

Example:

```
str r0, [r1], r2      @  *r1 ← r0   then   r1 ← r1 + r2
```

ARM Addressing Modes (Post-Indexing)

[R*n*], ±R*m*, *shift* *Scaled register post-indexed*

Address accessed is value found in R*n*, and then R*n*'s value is increased/decreased by R*m* shifted according to *shift*.

Example:

```
ldr r0, [r1], r2, lsl #3   @  r0 ← *r1   then   r1 ← r1 + r2*8
```

Examples of pre- and post- indexed addressing

```
str r3, [r0, r4, lsl #3]        @  pre-indexed
ldr r5, [r0, r1, lsl #3]!       @  pre indexed with writeback
ldr r0, [r1, #-8]               @  pre-indexed with negative offset
ldr r0, [r1, -r2, lsl #2]       @  negative offset shifted
ldrb r5, [r1]                   @  load byte from ea <r1>
ldrsh r5, [r3]                  @  load signed halfword from ea <r3>
ldrsb r5, [r3, #0xc1]           @  load signed byte from ea <r3+193>

str r7, [r0], #4                @  store r7 to ea<r0>, then add #24 to r0
ldr r2, [r0], r4, lsl #2        @  load r2 from ea<r0>, then add r4*4 to r0
ldrh r3, [r5], #2               @  load halfword to r3 from ea<r5>,  then
                                @  add #2 to r5
strh r2, [r5], #8               @  store halfword from r2 to ea<r5>, then
                                @  add 8 to r5
```

Block Data (Multiple data) transfer instructions

- ARM also supports multiple loads and stores:
- When the data to be copied to the stack is known to be a multiple of 4 bytes, copying is faster using load multiple and store multiple instructions.
- Each instruction can transfer 1 or more 32-bit words between registers and memory
- Each instruction updates another register that holds the memory address

Multiple data transfer instructions

General syntax:

op<address-mode>{cond} <rn>{!}, <register-list>{^}

- op : ldm, stm
- address-mode:

ia – Increment address after each transfer
ib – Increment address before each transfer.
da – Decrement address after each transfer
db – Decrement address before each transfer
fd – full descending stack
ed – empty descending stack
fa – full ascending stack
ea – empty ascending stack.

Multiple data transfer instructions

- cond is an optional condition code

- *rn* is the *base register* containing the initial memory address for the transfer.
- ! is an optional suffix.
  - If ! is present, the final address is written back into *rn*.
  - If the base register is in the register-list, then you must not use the writeback option.

Multiple data transfer instructions

reg-list

- a list of registers to be loaded or stored.
- can be a comma-separated list or an rx-ry range.
- may contain any or all of r0 - r15
- the registers are always loaded in order, regardless to how the registers are ordered in the list.
- for both the ldm and stm instructions, reg-list must not contain the sp
- for ldm, reg-list must not contain the PC if it contains the lr
- for stm, reg-list must not contain the lr if it contains the pc

Multiple data transfer instructions

- ^ is an optional suffix. Do NOT use it in User mode or System mode.
  - forces processor to transfer the saved program status register (SPSR) into the current program status register (CPSR) at the same time, saving us an instruction
  - if *op* is LDM and *register-list* contains the pc, the CPSR is restored from the SPSR
  - otherwise, data is transferred into or out of the User mode registers instead of the current mode registers.

Multiple data transfer instructions

Instruction	Operation	[s]	Notes
ldmia r_n[!],<reglist>	r_{first}<- mem[r_n] $r_{first+1}$<- mem[r_n+4] $r_{first+2}$<- mem[r_n+8] etc.		r_{first}, r_{last} … refer to the order of the registers by register number, not by the order they apprear in the list '!' is an optional write-back suffix. If present, the final address that is written to or stored from is written back into r_n
ldmdb r_{last}[!],<reglist>	r_{last}<- mem[r_n] r_{last-1}<- mem[r_n- 4] r_{last-2}<- mem[r_n- 8] etc.		
stmia r_n[!],<reglist>	mem[r_n] <- r_{first} mem[r_n+4] <- $r_{first+1}$ mem[r_n+8] <- $r_{first+2}$ etc.		
stmdb r_n[!],<reglist>	mem[r_n] <- r_{lasst} mem[r_n- 4] <- r_{last-1} mem[r_n- 8] <- r_{last-2} etc.		

Multiple data transfer instructions

Example of ldmia – load, increment after

```
ldmia  r9, {r0-r3}  // register 9 holds the
                    // base address. “ia” says
                    // increment the base addr
                    // after each value has
                    // been loaded from memory
```

Multiple data transfer instructions

Example of ldmia – load, increment after

```
ldmia  r9, {r0-r3}  // register 9 holds the
                    // base address
```

This has the same effect as four separate ldr instructions, or

```
ldr   r0, [r9]
ldr   r1, [r9, #4]
ldr   r2, [r9, #8]
ldr   r3, [r9, #12]
```

Note: at the end of the `ldmia` instruction, register r9 has not been changed. If you wanted to change r9, you could simply use

```
ldmia r9!, {r0-r3, r12}
```

Multiple register data transfer instuctions

ldmia – Example 2

```
ldmia r9, {r0-r3, r12}
```

- Load words addressed by r9 into r0, r1, r2, r3, and r12
- Increment r9 after each load.

Example 3

```
ldmia r9, {r5, r3, r0-r2, r14}
```

- load words addressed by r9 into registers r0, r1, r2, r3, r5, and r14.
- Increment r9 after each load.

- ldmib, ldmda, ldmdb work similar to ldmia
- Stores work in an analogous manner to load instructions

PUSH and POP

Note:
push is a synonym for stmdb sp!, reg-list

pop is a synonym for ldmia sp!, reg-list

Note:
ldmfd is a synonym for ldmia

stmfd is a synonym for stmdb

Multiple register data transfer instuctions

Common usage of multiple data transfer instructions

- Stack
  - Function calls
  - Context switches
  - Exception handlers

Multiple register data transfer instructions

Stack

- When making nested subroutine calls, we need to store the current state of the processor.
- The multiple data transfer instructions provide a mechanism for storing state on the *runtime stack* (pointed to by the stack pointer, r13 or sp)

stack addressing:

- stacks can *ascend* or *descend* memory
- stacks can be *full* or *empty*
- ARM multiple register transfers support all forms of the stack

Multiple register data transfer instructions

Stack

- Ascending stack: grows up
- Descending stack: grows down

A stack pointer (sp) holds the address of the current top of the stack

Full stack: sp is pointing to the last valid data item pushed onto the stack

Empty stack: sp is pointing to the vacant slot where the next data item will be placed

Multiple register data transfer instructions

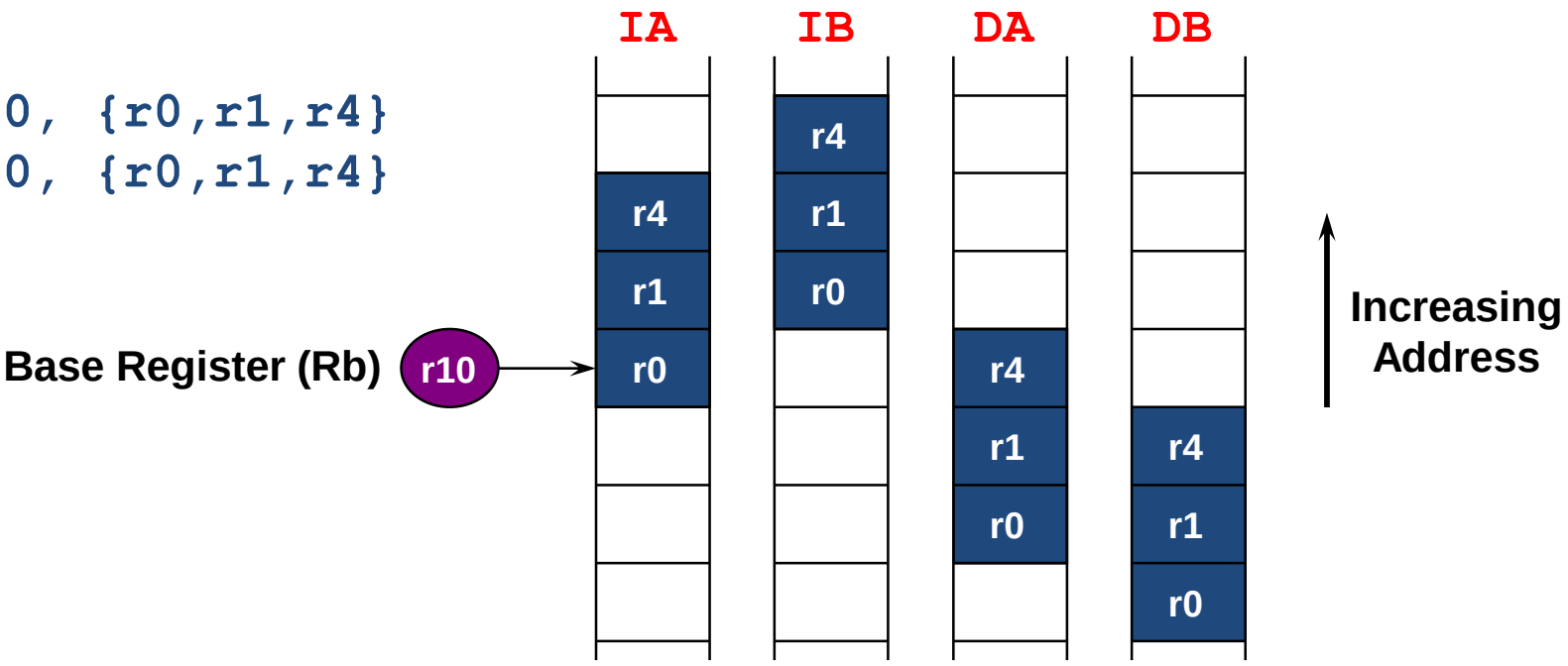
Stack Processing

ARM support for all four forms of stacks

- *Full ascending (FA)*: grows up; stack pointer points to the highest address containing a valid data item
- *Empty ascending (EA)*: grows up; stack pointer points to the first empty location
- *Full descending (FD)*: grows down; stack pointer points to the lowest address containing a valid data item
- *Empty descending (ED)*: grows down; stack pointer points to the first empty location below the stack

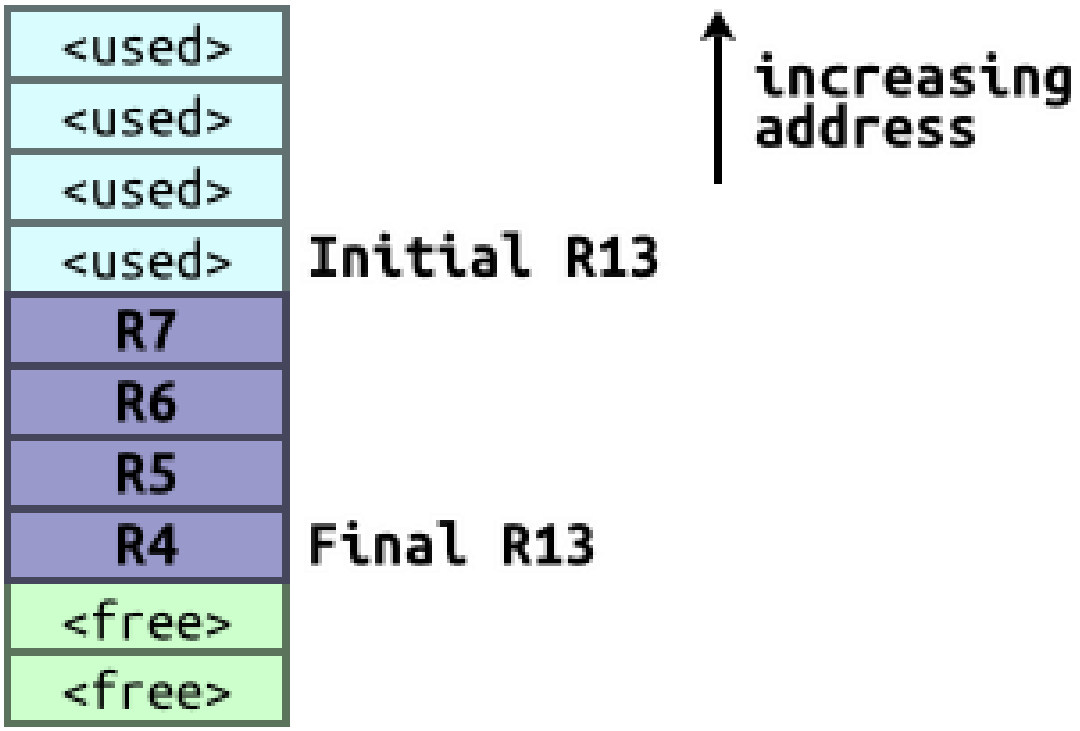
Load and Store Multiples

```
LDMxx r10, {r0,r1,r4}
STMxx r10, {r0,r1,r4}
```



Stack push operation: stmfd

STMFD r13!, {r4-r7} – Push R4,R5,R6 and R7 onto the stack.



Stack pop operation: ldmed

LDMFD r13!, {r4-r7} – Pop R4,R5,R6 and R7 from the stack.