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

ARM also supports multiple loads and stores:

General syntax:

\[
\text{op}<\text{address-mode}>{{\cond}}<\rn>{!}, <\text{register-list}>{^}\]

- 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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>[s]</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldmia ( r_n[!], &lt;\text{reglist}&gt; )</td>
<td>( r_{\text{first}} \leftarrow \text{mem}[r_n] )&lt;br&gt;( r_{\text{first}+1} \leftarrow \text{mem}[r_n+4] )&lt;br&gt;( r_{\text{first}+2} \leftarrow \text{mem}[r_n+8] )&lt;br&gt; etc.</td>
<td></td>
<td>( r_{\text{first}} ), ( r_{\text{last}} ) ... refer to the order of the registers by register number, not by the order of appearance in the list</td>
</tr>
<tr>
<td>ldmia ( r_{\text{last}}[!], &lt;\text{reglist}&gt; )</td>
<td>( r_{\text{last}} \leftarrow \text{mem}[r_n] )&lt;br&gt;( r_{\text{last}-1} \leftarrow \text{mem}[r_n-4] )&lt;br&gt;( r_{\text{last}-2} \leftarrow \text{mem}[r_n-8] )&lt;br&gt; etc.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stmia ( r_n[!], &lt;\text{reglist}&gt; )</td>
<td>( \text{mem}[r_n] \leftarrow r_{\text{first}} )&lt;br&gt;( \text{mem}[r_n+4] \leftarrow r_{\text{first}+1} )&lt;br&gt;( \text{mem}[r_n+8] \leftarrow r_{\text{first}+2} )&lt;br&gt; etc.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 instructions

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 instructions

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}

Base Register (Rb) r10

Increasing Address
Stack -- Last in first out memory

- Multiple store / load
  - `stmed`
  - `ldmed`

**Stack example**

<table>
<thead>
<tr>
<th>Address (H)</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000 0488</td>
<td>:</td>
</tr>
<tr>
<td>4000 0008</td>
<td>:</td>
</tr>
<tr>
<td>4000 0004</td>
<td>:</td>
</tr>
<tr>
<td>4000 0000</td>
<td>:</td>
</tr>
</tbody>
</table>
Stack push operation: stmed

Store multiple empty descending instruction

subr1:

stmed r13!, {r0-r2, r14}  @ push work & link registers
@ stores data on stack and decreases r13

STMEDr13!, {r0-r2, r14}
LDMED r13!, {r0-r2, r14}

“Empty” means Stack Pointer is pointing to an empty location
Stack pop operation: ldmed

Load multiple empty descending

ldmed  r13!, {r0-r2, r14}  @ pop work & link registers
@ restores data to registers
@ and increases r13

STMEDr13!, {r0-r2, r14}

when return from SUB1

(r14)
(r2)
(r1)
(r0)

LDMED  r13!, {r0-r2, r14}
Stack push operation: stmfd

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