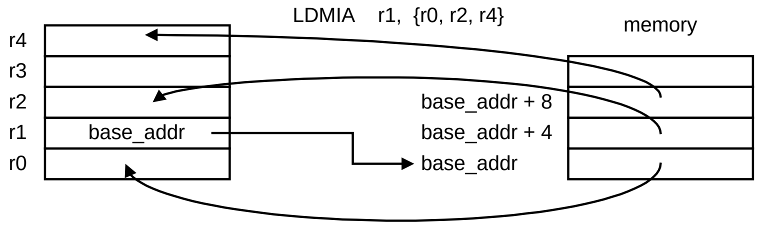
Lecture 7 Stacks and Subroutines

- LDR and STR instructions only load/store a single 32-bit word.
- ARM can load/store ANY subset of the 16 registers in a single instruction. For example:

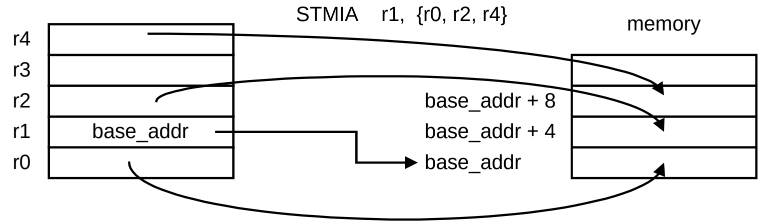
```
LDMIA   r1, {r0, r2, r4}        ; r0 := mem32[r1]
                                ; r2 := mem32[r1+4]
                                ; r4 := mem32[r1+8]
```



Load/Store Multiple Instructions

- Any registers can be specified. However, beware that if you include r15 (PC), you are effectively forcing a branch in the program flow.
- The complementary instruction to LDMIA is the STMIA instruction:

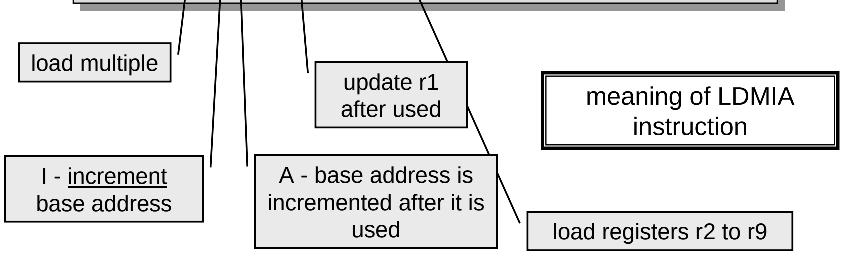
```
STMIA   r1, {r0, r2, r4}        ; mem32[r1] := r0
                                ; mem32[r1 + 4] := r2
                                ; mem32[r1 + 8] := r4
```



Update base address register with Load/Store Multiple Instructions

- So far, r1, the base address register, has not been changed. You can update this pointer register by adding '!' after it:

```
LDMIA   r1 ! , {r2-r9}          ; r2 := mem32[r1]
                                ;   ...............
                                ; r9 := mem32[r1+28]
                                ; r1 := r1 + 32
```



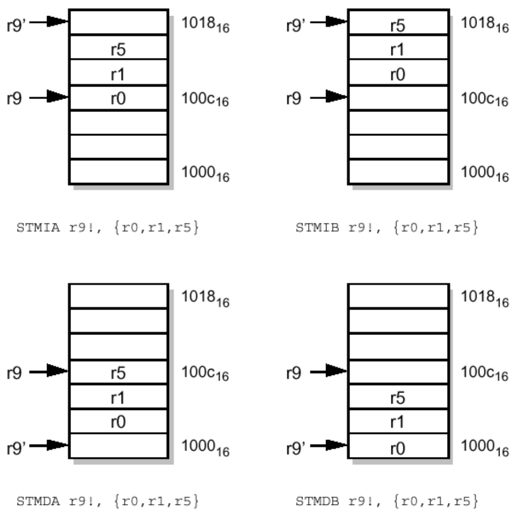
Example of using Load/Store Multiple

- Here is an example to move 8 words from a source memory location to a destination memory location:-

```
ADR     r0, src_addr    ; initialize src addr
ADR     r1, dest_addr   ; initialize dest addr
LDMIA   r0!, {r2-r9}    ; fetch 8 words from mem
                        ;   ... and update r0 := r0 + 32
STMIA   r1, {r2-r9}     ; copy 8 words to mem, r1 unchanged
```

- When using LDMIA and STMIA instructions, you:-
  - **INCREMENT** the address in memory to load/store your data
  - the increment of the address occurs **AFTER** the address is used.
- In fact, one could use 4 different form of load/store:
  - **I**ncrement - **A**fter LDM**IA** and STM**IA**
  - **I**ncrement - **B**efore LDM**IB** and STM**IB**
  - **D**ecrement - **A**fter LDM**DA** and STM**DA**
  - **D**ecrement - **B**efore LDM**DB** and STM**DB**

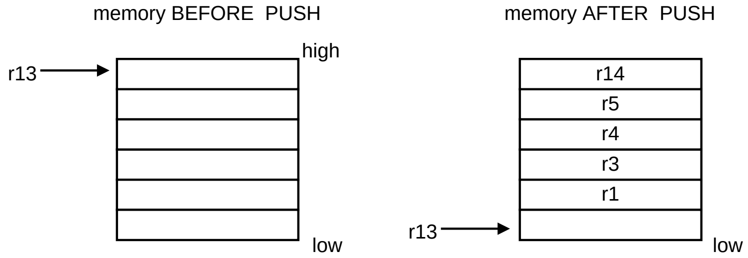
The four variations of the STM instruction



The idea of a STACK

- The multiple load/store instructions can be used to implement **last-in-first-out storage** called a **STACK**.
- A stack is a portion of main memory used to store data temporarily
- A PUSH operation which stores a number of registers onto the stack memory.

PUSH {r1, r3-r5, r14}



PUSHing onto a Stack

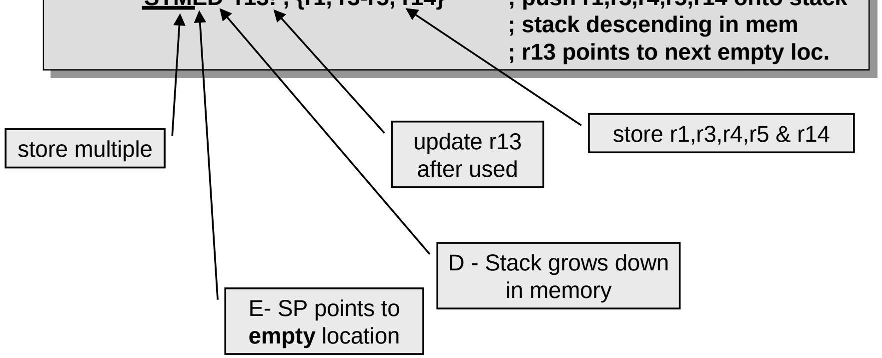
- Note the following properties of the PUSH operation:
  - **r13** is used as the address pointer. We call this **STACK POINTER (SP)**. We could have used any other registers (except r15) as SP, but it is good practice to use **r13** unless there is a good reason not to do so.
  - The stack grows **down** through **decreasing** memory address, and
  - The base registers points to the first **empty** location of the stack. To store values in memory, the SP is **decremented after** it is used.
- ARM does not have a PUSH instruction, but we can use one of the STM instructions to implement a PUSH operation.
- Consider page 6-5, it is clear that we can implement PUSH as described with a STMDA instruction:

```
STMDA r13!, {r1, r3-r5, r14}     ; Push r1, r3-r5, r14 onto stack
                                 ; Stack grows down in mem
                                 ; r13 points to next empty loc.
```


Stack view of STM instructions

- In ARM terminology, STM instruction used to implement a stack can have a different name. The STMDA instruction as we have seen is equivalent to a **STMED** instruction:

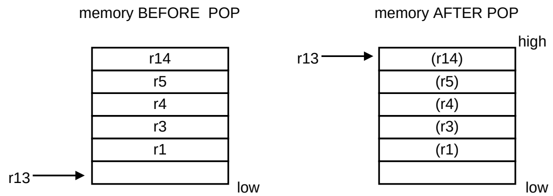
```
STMED r13! , {r1, r3-r5, r14}    ; push r1,r3,r4,r5,r14 onto stack
                                 ; stack descending in mem
                                 ; r13 points to next empty loc.
```



POP operation

- The complementary operation of PUSH is the **POP** operation.

POP {r1, r3-r5, r14}



- ARM does not have a POP instruction. In this case, we can use:

```
LDMIB   r13!, {r1, r3-r5, r14}          ; Pop r1, r3-r5, r14 from stack
```

- This is equivalent to the stack manipulation instruction:

```
LDMED  r13!, {r1, r3-r5, r14}          ; Pop r1, r3-r5, r14 from stack
```


The four different ways of implementing a stack

- **Ascending/Descending**: A stack is able to grow upwards, starting from a low address and progressing to a higher address—an ascending stack, or downwards, starting from a high address and progressing to a lower one—a descending stack.
- **Full/Empty**: The stack pointer can either point to the top item in the stack (a full stack), or the next free space on the stack (an empty stack).

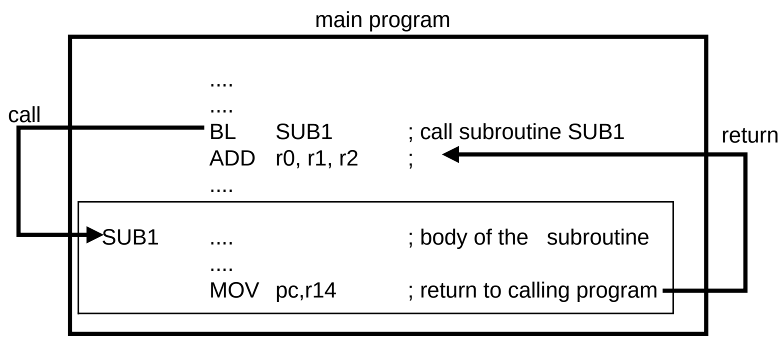
```
STMFA  r13!, {r0-r5}; Push onto a Full Ascending Stack
LDMFA  r13!, {r0-r5}; Pop from a Full Ascending Stack
STMFD  r13!, {r0-r5}; Push onto a Full Descending Stack
LDMFD  r13!, {r0-r5}; Pop from a Full Descending Stack
STMEA  r13!, {r0-r5}; Push onto an Empty Ascending Stack
LDMEA  r13!, {r0-r5}; Pop from an Empty Ascending Stack
STMED  r13!, {r0-r5}; Push onto Empty Descending Stack
LDMED  r13!, {r0-r5}; Pop from an Empty Descending Stack
```


Relationship between the two different views of LDM/STM instructions

Name	Stack	Other
pre-increment load	LDMED	LDMIB
post-increment load	LDMFD	LDMIA
pre-decrement load	LDMEA	LDMDB
post-decrement load	LDMFA	LDMDA
pre-increment store	STMFA	STMIB
post-increment store	STMEA	STMIA
pre-decrement store	STMFD	STMDB
post-decrement store	STMED	STMDA

Subroutines

- Subroutines allow you to modularize your code so that they are more reusable.
- The general structure of a subroutine in a program is:



Subroutine (con't)

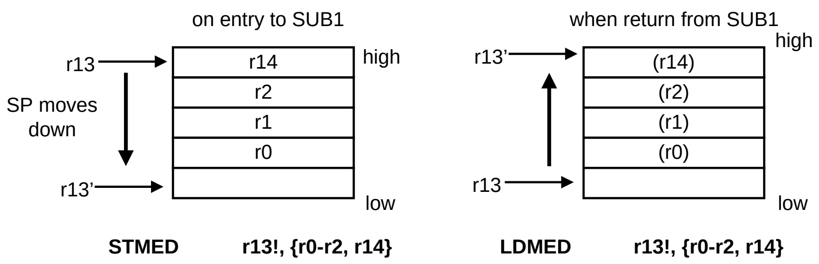
- `BL    subroutine_name` (Branch-and-Link) is the instruction to jump to subroutine. It performs the following operations:
  - 1) It saves the **PC** value (which points to the next instruction) in r14. This is the return address.
  - 2) It loads **PC** with the address of the subroutine. This performs a branch.
- BL always uses r14 to store the return address. r14 is called the **link register** (can be referred to as lr or r14).
- Return from subroutine is simple: - just put r14 back into PC (r15).

Nested Subroutines

- Since the return address is held in register r14, you should not call a further subroutine without first saving r14.
- It is also a good software engineering practice that a subroutine does not change any register values except when passing results back to the calling program.
- This is the principle of information hiding: try to hide what the subroutine does from the calling program.
- How do you achieve these two goals? Use a stack to:
  - Preserve r14
  - Save, then retrieve, the values of registers used inside subroutine

Preserve things inside subroutine with STACK

```
        BL      SUB1
        .....
SUB1    STMED   r13!, {r0-r2, r14}        ; push work & link registers
        ....
        BL      SUB2                      ; jump to a nested subroutine
        ...
        LDMED   r13!, {r0-r2, r14}        ; pop work & link registers
        MOV     pc, r14                   ; return to calling program
```



Effect of subroutine nesting

- SUB1 calls another subroutine SUB2. Assuming that SUB2 also saves its link register (r14) and its working registers on the stack, a snap-shot of the stack will look like:-