Load/Store Instructions

- ARM is a RISC architecture
  - which means a Load/Store architecture
  - does **not allow** computations directly from memory
  - requires load/store between registers and memory for ALU computations
- Load/Store instructions in ARM
  - LDR / STR : single load/store
  - LDM / STM : multiple load/store
  - SWP : data swap

Single Load / Store

- Instructions

Access Unit	Load inst	Store inst
Word	LDR	STR
Byte	LDRB	STRB
Halfword	LDRH	STRH
Signed byte	LDRSB	
Signed halfword	LDRSH	

  - 단위
    - Word : 32 bits
      - CPU가 한 클락에 처리할 수 있는 데이터의 단위
    - Byte : 8 bits
    - Halfword : 16 bits
      - Byte, Halfword는 많이 쓰이진 않음
  - Align 된 상태로 명령어가 작성되길 기대
    - 잘 안되면 Exception 발생 (Data abort 등)
  - Signed 명령은 왜 있을까?
- Conditional executions are possible
  - ex) LDR**EQ**

LDR Instruction

- Format
  - `LDR {cond} {size} dest, <address>`
    - `dest` is the destination register
    - `<address>` is the target address with `BASE:OFFSET` format
    - Reads at the amount of `size` from the memory with `address`
- Usages
  - `LDR R1, [R2, R4]`
    - Loads from the address `[R2+R4]` to Register `R1`

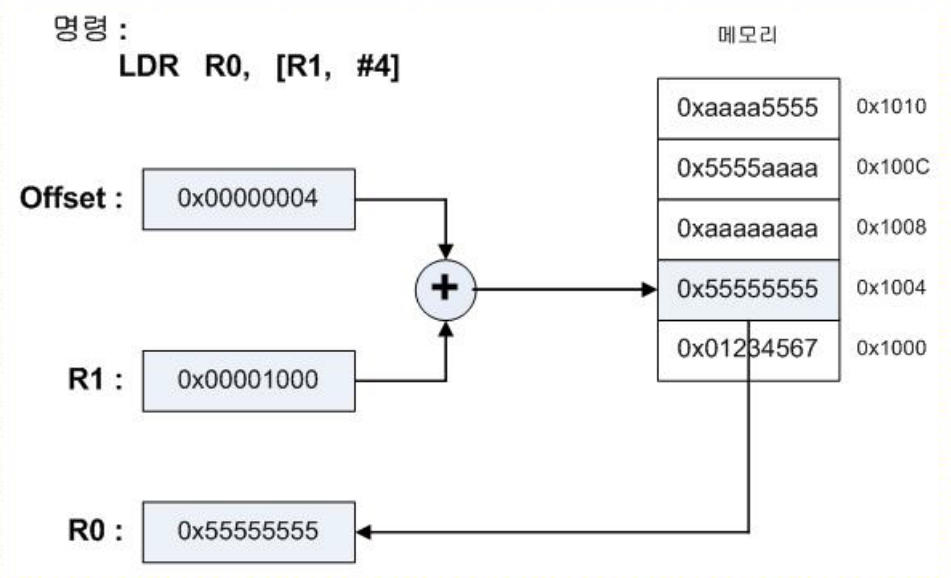
        - `R2 = BASE`, `R4 = OFFSET`
    - `LDREQB R1, [R6, #5]`
        - Loads a **byte data** from the address `[R6+5]` to register `R1` only when the previous ALU computation sets **Z bit** of condition flags

STR Instruction

- Format
    - `STR {cond} {size} src, <address>`
        - `src` is the source register
        - `<address>` is the target address with `BASE:OFFSET` format
        - Stores the amount of `size` from Register `src` to the memory with `address`
- Usages
    - `STR R1, [R2, R4]`
        - Stores from Register R1 to the address `[R2+R4]`
    - `STREQB R1, [R6, #5]`
        - Stores a byte data from Register R1 to the address `[R6+5]` only when the previous ALU computation sets Z bit of condition Flags

Addressing Modes in Load / Store

- Pre-Indexed



    - The target address is **calculated first before actual accessing the address**
        - The value of base register is not changed after data access if auto-update is not set
    - 먼저 계산해서 source address를 만들어냄
        - `[Rn, Offset]{!}`
        - `!` means **auto-update**
    - `LDR R0 [R1, R4]{!}`
        - `R1`이 0x10, `R4`가 0x04일 때
        - `R1`은 0x14가 되고, `R4`는 그대로

```
; template code: store 0x11, 0x22, 0x33, 0x44 to mem[0x1000 ~0x100c]
mov     R11, #0x1000  ; base addr
mov     R12, #0x11
str     R12, [R11, #0x0]
mov     R12, #0x22
str     R12, [R11, #0x4]
mov     R12, #0x33
str     R12, [R11, #0x8]
mov     R12, #0x44
str     R12, [R11, #0xC]
```

```
;LDR        preindexed:
;use        r4 as a base register
;use        immediate offset
;destination    registers: r0 to r3

mov     R4, #0x1000 ; base
ldr     R0, [R4, #0x0]
ldr     R1, [R4, #0x4]
ldr     R2, [R4, #0x8]
ldr     R3, [R4, #0xC]
```

```
;LDR        preindexed with auto update:
;use        r4 as a base register and r5 as an offset register
;use        auto update mode only
;destination    registers: r0 to r3

mov     R4, #0x1000 ; base
mov     R5, #0x4 ; offset
sub     R4, R4, R5 ; 이 명령이 없으면 0x1004로 시작해버린다
ldr     R0, [R4, R5]!
ldr     R1, [R4, R5]!
ldr     R2, [R4, R5]!
ldr     R3, [R4, R5]!
```

- Post-indexed
  - The target address is calculated / modified after the data access for load/store has been performed
  - 일단 Access를 하고(명령을 하고), 주소를 바꾼다
    - `[Rn], Offset`

```
;LDR        post indexed:
;use        r4 as a base register and r5 as an offset register
;use        post indexed mode only
;destination    registers: r0 to r3

mov     R4, #0x1000 ; base
mov     R5, #0x4 ; offset
ldr     R0, [R4], R5
ldr     R1, [R4], R5
ldr     R2, [R4], R5
ldr     R3, [R4], R5
```

PC-relative Addressing Mode

- LABEL-based addressing
  - If LABEL is used in assembly code, that is transformed to [PC+LABEL] in machine code

    - 용도 : 해당 Constant Number를 쓰고싶다(MOV로 해결 안되는 절대 주소 등)
    - 없는 명령이지만, 어셈블러가 어셈블링을 할 때 잘 번역해서 이해함
    - LDR로부터 해당 명령이 얼마나 떨어졌는지 확인하고, 그만큼 이동시킴(어셈블러가 계산)
- Literal pool (Data) addressing

  - If an address value is used as assignment form in assembly code, that is transformed to [PC+LABEL] with the address value is stored in data area at LABEL
    - 그냥 value만 쓰면 알아서 LABEL 형식으로도 만들어줌(어셈블러가)

Block Transfer Load/Store

- Block data transfer instructions
  - Load/Store multiple data with a single instruction
  - LDM(Load Multiple), STM(Store Multiple)
- Block data transfer examples
  - Memory copy, array move, and so on
  - Stack operations
    - ARM does not have stack instructions(Push / Pop)
    - LDM/STM instructions are used to implement stack operations

LDM and STM Formats

- 데이터 offset의 단위 : Word(4 byte, 32 bit) 고정
- Load multiple
  - `LDM{cond} <address mode> base_register{!}, <register list>`
  - Loads multiple data from the address in `base_register` to registers listed in `<register list>`
  - increment/decrement 여부(I/D)
  - increment 후 word를 access할지(B), word access 후 increment할지 지정(A)
  - !가 들어가면 base_register update 여부
  - ex) `LDMIA R0, {R1, R2, R3}`
- Store multiple
  - `STM{cond} <address mode> base_register{!}, <register list>`
  - Stores multiple data from registers listed in `<register list>` to memory area starting from `base_register`
  - ex) `STMIA R0, {R1, R2, R3}`

- Register list
  - `{R1, R2, R3}` or `{R0-R5}`
  - From R0 to R15 (PC) can be used(최대 16개)

  - 순서를 나타낼 방법이 없다?
    - 어떤 순서로 집어넣던지, 실제로 하는일은 똑같다(16개의 비트를 이용해 체크하는 방식)
    - 집합으로 생각하는 것이 맞다

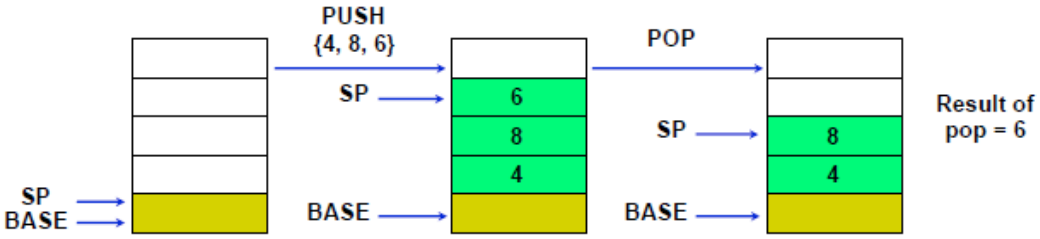
Addressing Modes

Addressing Mode	Insts		Address calculation
	Regular	Stack	
Pre-increment Load	LDMIB	LDMED	Increment before load
Post-increment Load	LDMIA	LDMFD	Increment after load
Pre-decrement Load	LDMDB	LDMEA	Decrement before load
Post-decrement Load	LDMDA	LDMFA	Decrement after load
Pre-increment Store	STMIB	STMFA	Increment before store
Post-increment Store	STMIA	STMEA	Increment after store
Pre-decrement Store	STMDB	STMFD	Decrement before store
Post-decrement Store	STMDA	STMED	Decrement after store

- 저장할때 Decrement 방식이였으면, 불러올때는 Increment로 불러와야함(반대도 마찬가지)
- 저장할때 After 방식이였으면, 불러올때는 Before방식으로(반대도 마찬가지)
- Stack
  - Empty(After) / Full(Before)
  - Ascending(Increment) / Descending(Decrement)

Stack Operations

- Push and pop
  - push stores a data on the top of the stack while pop obtains the data on the top of the stack and eliminates the data from the stack
    - Stack pointer is the top location of the stack
    - base pointer is the bottom location of the stack

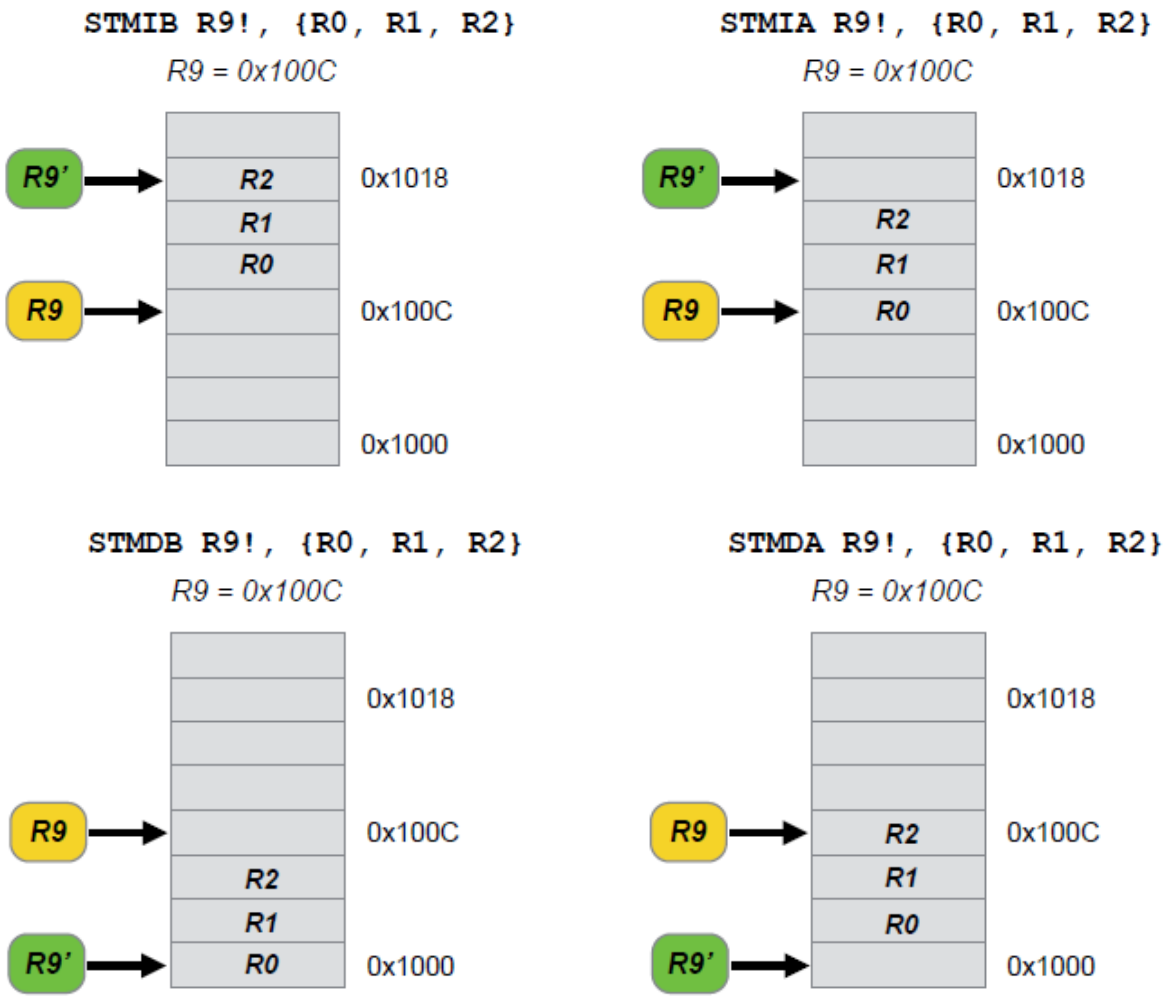


Stack Types

- Full stack
  - the location of the stack pointer is filled with the data stored by the last push operation
  - the next push needs decrease / increase of the stack pointer
  - FD / FA stacks
- Empty stack
  - the location of the stack pointer is empty and thus the next push operation stores the data to the stack pointer location
  - after push operation, the stack pointer is increased / decreased to point to another empty location
  - ED / EA stacks

Examples

- 메모리 주소가 낮은쪽을 밑으로 하는 것이 ARM 국룰
  - R0가 제일 밑으로, R15가 제일 위로 가도록



- LDM

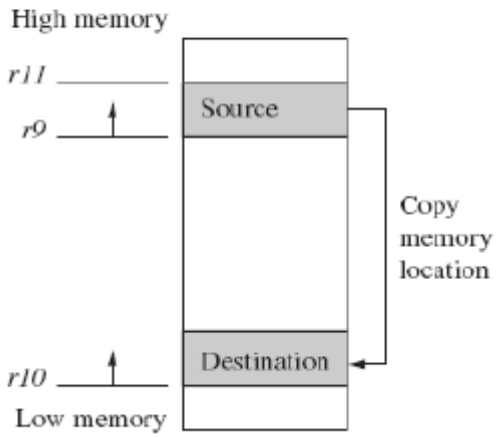
```
PRE     mem32[0x80018] = 0x03
        mem32[0x80014] = 0x02
        mem32[0x80010] = 0x01
        r0 = 0x00080010
        r1 = 0x00000000
        r2 = 0x00000000
        r3 = 0x00000000

LDMIA r0!, {r1-r3}

POST    r0 = 0x0008001c
        r1 = 0x00000001
        r2 = 0x00000002
        r3 = 0x00000003
```

- Memory copy

```
loop
LDMIA r9!, {r0-r7}
STMIA r10!, {r0-r7} ; and store them
CMP r9, r11
BNE loop
```



  - 8개씩 하는게(spatial locality) 효율이 좋다
  - 데이터가 끝날때까지(r11의 위치) 반복