Load / Stroe Instrucitons

- ARM은 RISC 아키택처
  - Load / Store 아키텍처를 의미함
  - 메모리에서 직접 연산을 허용하지 않음

    = memory to memory 데이터 처리 명령을 지원하지 않음

    = 처리하고자 하는 데이터는 무조건 레지스터로 이동해야함

    - 처리절차

      1. 데이터를 메모리에서 레지스터로 이동
      2. 레지스터에 읽혀진 값을 가지고 처리
      3. 연산 결과를 메모리에 저장
  - ALU 계산을 위해 레지스터와 메모리 사이에 Load / Store 필요
- ARM에서 Load / Stroe 명령어
  - LDR / STR : 단일 load / store , 단일 레지스터 데이터 전송 명령
  - LDM / STM : 다중 load / store, 블록 단위 데이터 전송 명령
  - SWP : 데이터 스왑, 단일 레지스터를 사용한 메모리와 레지스터 내용을 스왑

Single Load / Store

- 단일(single) load / store 명령어

억세스 단위	Load 명령	Store 명령
워드(Word)	LDR	STR
바이트(Byte)	LDRB	STRB
하프워드(Halfword)	LDRH	STRH
Signed 바이트	LDRSB	
Signed 하프워드	LDRSH	

- 모든 명령어는 STR / LDR 다음에 적절한 condition 코드를 삽입하여 조건부 실행 가능
  - ex) LDREQ
- Aligement 안맞으면 -> exception (Data)
- 왜 signed가 따로있을까? 그 자료형이 sigend로 선언되어있기 때문에, 산술연산을 정상적으로 하려면 필요
  - unsigned : 32bit에 맞춰서 zero extends
  - signed : 32bit에 맞춰서 sign exteds

LDR Instruction

- LDR{cond}{size}{T} Rd, <Address>

  {cond}는 조건부 실행을 위한 condition 조건을 나타냄

  {size}는 전송되는 데이터 크기를 나타내며, B(byte), H(halfword), SB(signed byte), SH(signed halfword)

  {T}는 post-indexed 모드에서 non-privileged 모드로 데이터가 전송 (priviled mode = kernel mode)

Rd는 읽어온 메모리 값이 write되는 레지스터

<Address>부분은 베이스 Reg와 offest으로 구성 (OS를 쓴다는 가정하에!)

- <Address>가 나타내는 위치의 데이터를 {size} 만큼 읽어서 Rd에 저장한다.
- 사용 예
  - LDR R1, [R2, R4] R2(base) + R4(offset) 위치의 데이터를 R1에 저장
  - LDREQB R1, [R6, #5] 조건이 EQ이면(전 단계 ALU연산이 Z bit을 set하면) R6 + 5 위치의 데이터를 바이트 만큼 읽어 R1에 전달

STR Instruction

- STR{cond}{size}{T} Rd, <Address>

  {cond}는 조건부 실행을 위한 condition 조건을 나타냄

  {size}는 B(byte), H(halfword)

  {T}는 post-indexed 모드에서 non-privilege 모드로 데이터가 전송

  Rd는 write할 데이터가 들어있는 Reg

  <Address> 부분은 베이스 Reg와 Offest으로 구성
- <Address> 위치에 Rd를 저장
- 사용 예
  - STR R1, [R2, R4] R2 + R4 위치에 R1의 내용을 저장
  - STREQB R1, [R6, #5] 조건이 EQ이면 R6 + 5 위치에 R1을 byte 단위 저장

Addressing Modes in Load / Store

LDR R0, [R1, R4]

- Pre-Indexed
  - target 주소는 실제 주소에 접근하기 전에 먼저 계산됨
    - [Rn, Offset]{!}

```
        ;       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]

        mov     R4, #0x1000
        ldr     r0, [R4, #0x0]
        ldr     r1, [R4, #0x4]
        ldr     r2, [R4, #0x8]
        ldr     r3, [R4, #0xC]
```

    - '!' meas auto-update
      - use r4 as a base register and r5 as an offset register
      - use auto update mode only

    - destination registers: r0 to r3

```
        ;       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]

        mov     R4, #0x1000
        mov     R5, #0x04           ; 건너뛰는 term
        sub     R4, R4, #0x4        ; 한번 0x04 올라가므로 미리 빼줘야함
        ldr     r0, [R4, R5] !
        ldr     r1, [R4, R5] !
        ldr     r2, [R4, R5] !
        ldr     r3, [R4, R5] !
```

- Post-Indexed
  - Load/Store에 대한 데이터 접근이 수행된 후 target주소가 계산/수정됨

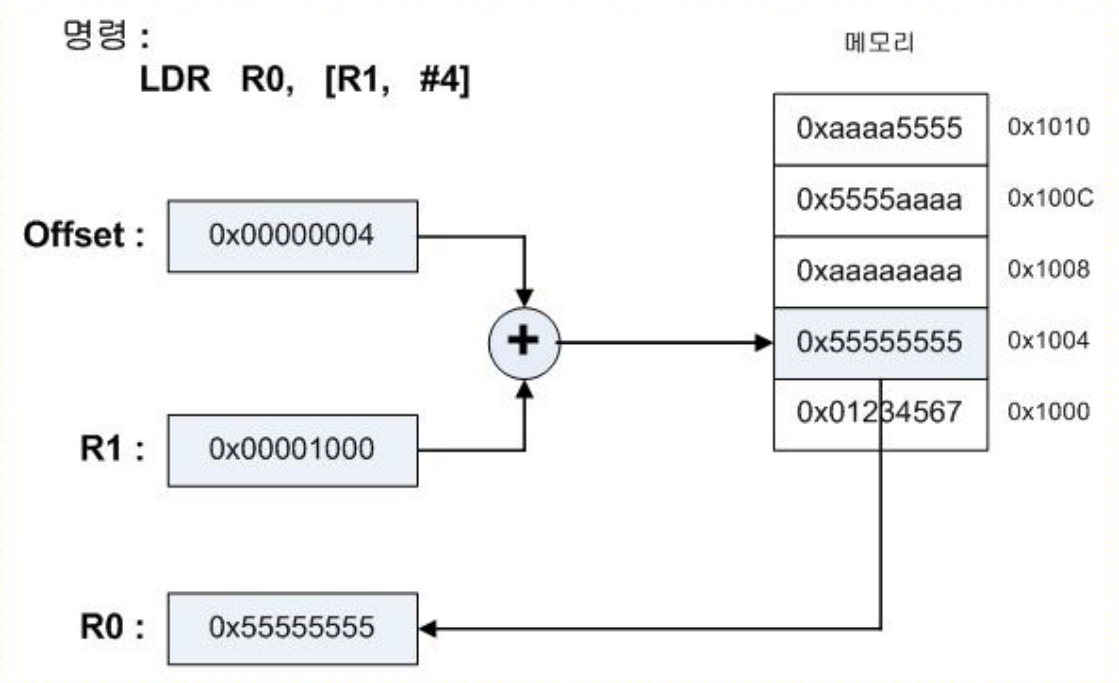
    = Base 값 만으로 명령어 실행 후 Base 값에 Offset을 더해 Base에 저장(다음 실행 시 반영)
    - [Rn], Offset
      - ex) LDR R0, [R1], #4 ; R0에 R1의 값을 저장 한 후 R1의 값을 그 다음 주소값으로 update

```
        ;       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]

        mov     R4, #0x1000
        mov     R5, #0x04       ; 0x04 이전 주소로 빼주는 거 안해도됨
        ldr     r0, [R4], R5    ; R4(0x1000)으로 먼저 실행되고 0x04 offset
후 update
        ldr     r1, [R4], R5
        ldr     r2, [R4], R5
        ldr     r3, [R4], R5
```

Pre-Indexed Addressing Mode

- target 주소 계산 후 데이터 접근 수행
  - auto-update가 설정되지 않은 경우 데이터 액세스 후 Base Reg 값이 변경되지 않음



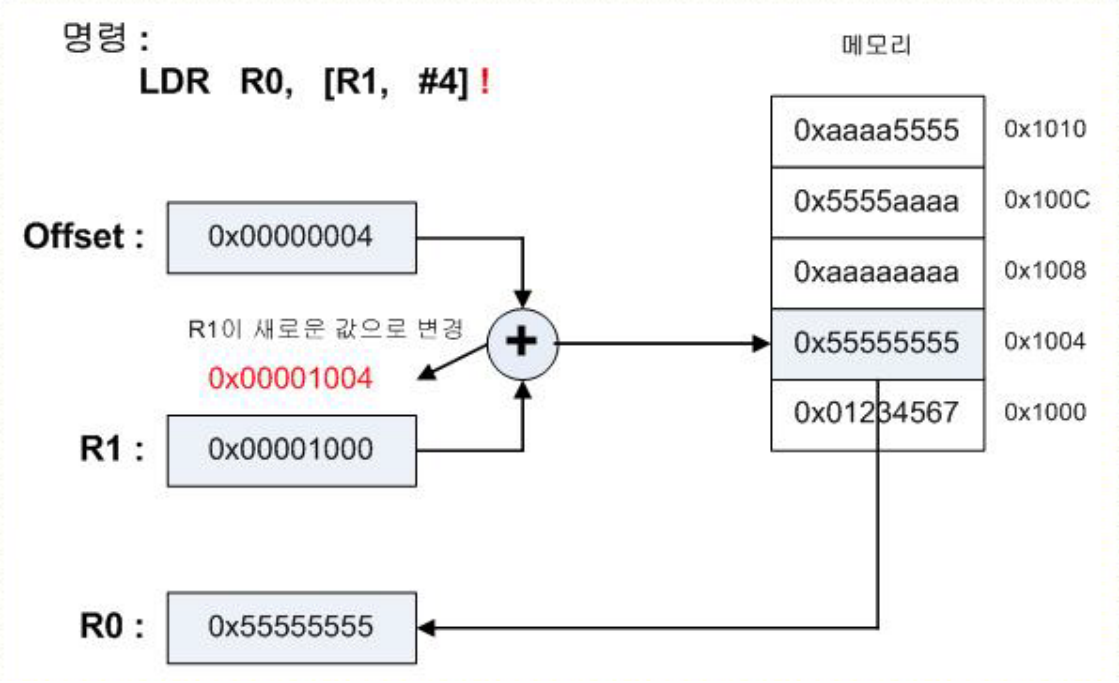
R1 : 0x00001000

R1 + #4 : 0x00001004

R0: 0x55555555

Pre-Indexed with Auto-Update

- 데이터 액세스 후 Base Reg update를 제외하고 pre-indexed addressing과 동일



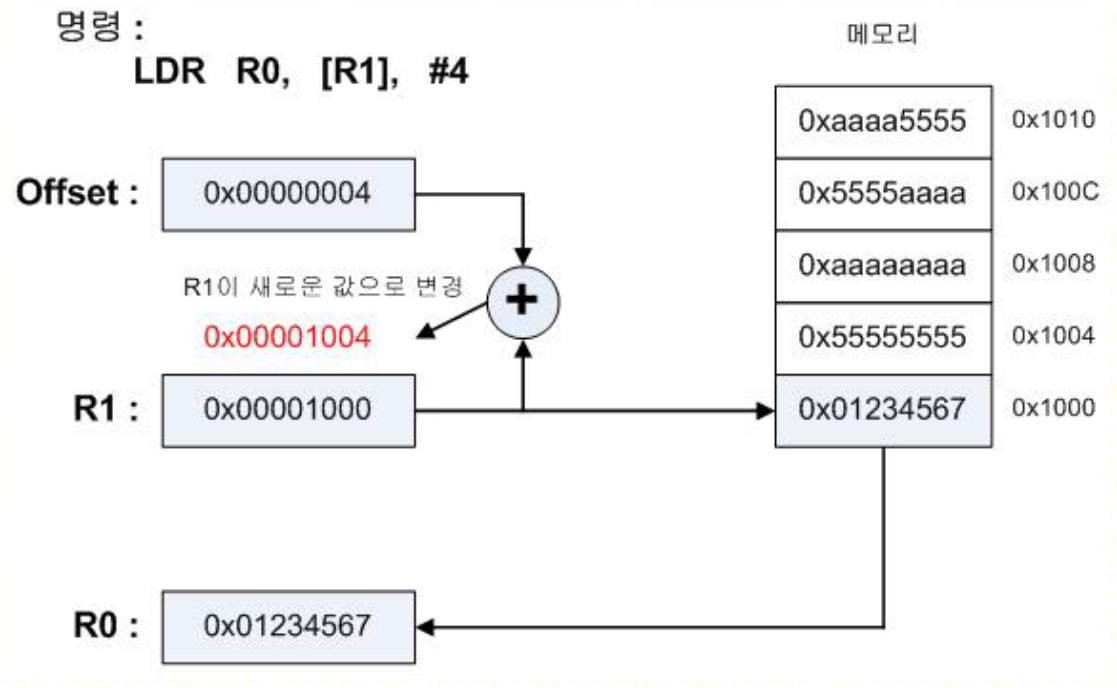
R1: 0x00001000 -> (auto-update) 0x00001004 -> (명령어 실행)

R1 + #4 : 0x00001004

R0: 0x55555555

Post-Indexed Addressing Mode

- Base Reg에 대한 데이터 액세스가 먼저 수행된 다음, Base Reg값이 Offset과 함께 새 값으로 업데이트됨
- Base Reg가 지정하는 주소에 데이터으이 전송 후 Rn값과 offset의 연산 결과를 Rn 갱신



R0 : 0x00001000 -> (명령어 실행) -> (update) 0x000010004

R0 + #4: 0x00001004

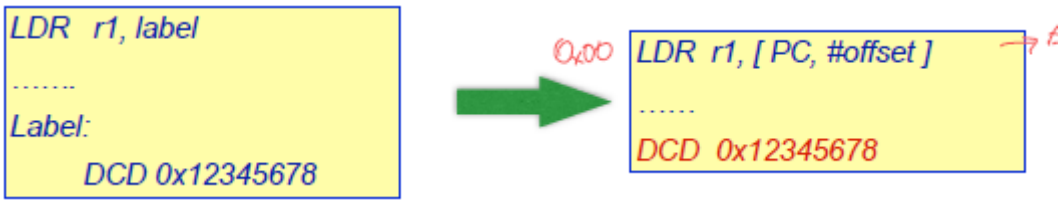
R1: 0x12345678

PC-relative Addressing Mode

label - 코드가 아닌데 코드에 포함

PC값에 상대적인 address : 컴파일 된 다음에 어셈블러가 상황에 따라서 주소 연속적으로 지정해줌

- PC 값에 명령어의 주소 필드의 값을 더해서 유효주소를 구하며, 분기 명령어 근처에 분기될 위치가 있는 경우 흔히 사용한다.
  - 직접 데이터에 접근할 수 없다.
  - 메모리를 효율적으로 사용할 수 있다.
  - 주소에 변위를 더해야만 한다.
  - 장 : 일반적인 분기 명령어보다 적은 수의 비트만 있으면 되는 것이 장점
- LABLE 기반 주소 지정
  - 어셈블리어에서 LABEL을 지정하면 어셈블러가 [PC + LABEL]형태의 주소로 변환하여 참조



- Literal pool (Data) addressing
- 어셈블러가 코드 영역 내에 데이터 저장 후 [PC + LABEL]현태의 주소로 변환하여 참고

메모리 엑세스가 추가돼서서 수행시간 더 길어짐

- 사용할 경우: 메모리 접근 시간과 compiler에서 같은 효과 처리를 위한 명령어를 더 추가하기 때문에, code의 크기도 늘어나고, 시간도 많이 듦 -> 꼭 필요할 때만
- 왜 필요? Branch 명령어의 경우 32bit 중 Branch 명령어 자체가 차지하는 9bit를 제외한 23bit를 branch하는 주소로 사용하는데, 이 23bit를 ARM 명령어로 생각해서 4byte당 한주소씩응로 따져 2비트 Left Shift를 사용해 봤자 분기 영역은 최대로 상수 표현 범위 (pc +- 32MB)라서 필요에 따라 그 이상의 분기가 필요할 때 32bit fully 사용 가능한 의사 명령어를 fuctionally 하게 이용

Block Transfer Load / Store

arm 에서는 push, pop 지원(x)

pre-indexed: Base에 Offset을 계산하고 실행

post-indexed: 실행하고, Base에 Offset을 계산

LDM/STM 은 offset을 지정 x, 암묵적으로 4byte

stack에서 push는 STM, pop 은 LDM

STMIB이면 LDMDA

- Block Data Transfer 명령
  - 하나의 명령으로 메모리와 프로세서 레지스터 사이에 여러개의 데이터를 옮기는 명령
  - Load 명령인 LDM과 Store 명령이 STM 명령
- Block Data Transfer 명령 으용
  - Memory copy, memory move 등
  - Stack operation
    - ARM에는 별도의 stack 관련 명령이 없다
    - LDM/STM 이용하여 pop 또는 push 동작 구현

LDM and STM formats

- LDM
  - LDM{cond}<addressing mode> Rn{!}, <register_list>

    {cond}는 조건부 실행을 위한 condition 조건을 나타냄

    {address mode}는 어드레스의 생성 방법을 정의

    Rn은 base 레지스터를 나타냄

    {!}는 데이터 전송 후 base 레지스터를 auto-update

    <register_list> 읽어온 데이터를 저장할 레지스터 지정
  - 동작: Base 레지스터 Rn이 지정한 번지에서 여러 개의 데이터를 읽어 <register_list>로 읽어들이는 명령
  - ex: LDMIA R0, {R1, R2, R3} ; R0의 위치에서 데이터를 읽어 R1, R2, R3에 저장
- STM
  - STM{cond}<addressing mode> Rn{!}, <register_list>

    {cond}는 조건부 실행을 위한 condition 조건을 나타낸다.

    {address mode}는 어드레스의 생성 방법을 정의한다.

    Rn은 base 레지스터를 나타낸다.

    {!}는 데이터 전송 후 base 레지스터를 auto-update

    <register_list> 읽어온 데이터를 저장할 레지스터 지정

- 동작: Base 레지스터 Rn이 지정한 번지에 <register_list>에 있는 여러 개의 데이터를 저장하는 명령
  - ex: STMIA R0, {R1, R2, R3} ; R1, R2, R3의 데이터를 R0의 위치주소부터 차례로 저장, 동작 완료 후 `R0는 변화 없음`.

    STMIA R0 !, {R1, R2, R3} ; R1, R2, R3의 데이터를 R0의 위치주소부터 차례로 저장, 동작 완료후 R0 = R0 + 12 (3 워드 주소값)로 변화

- <register_list>에서 사용가능한 레지스터

  R0에서 R15(PC)까지 최대 16ro

- 연속된 레지스터 표현

  {r0-r5}와 같이 "-"로 표현 가능

- <register_list>의 순서 지정

  항상 low order의 register에서 high order 순으로 지정

- XXX R9 !, {r0, r1, r5} 일때
  - Descending일 경우 r5부터 스택에 들어감
  - Ascending일 경우 r0부터 스택에 들어감

--> 어떤 방식으로 stack에 쌓던지 간에 r0는 낮은 주소에, r5는 높은 주소에 들어가게 된다.

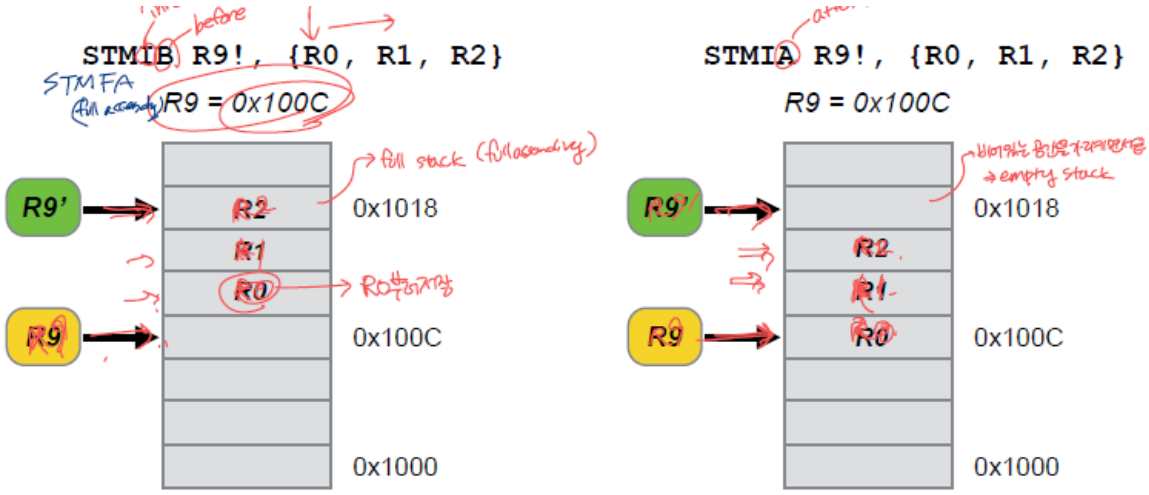
Addressing Modes in LDM / STM

Addressing Mode	키워드(표현방식)		유효 어드레스 계산
	데이터	스택	
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

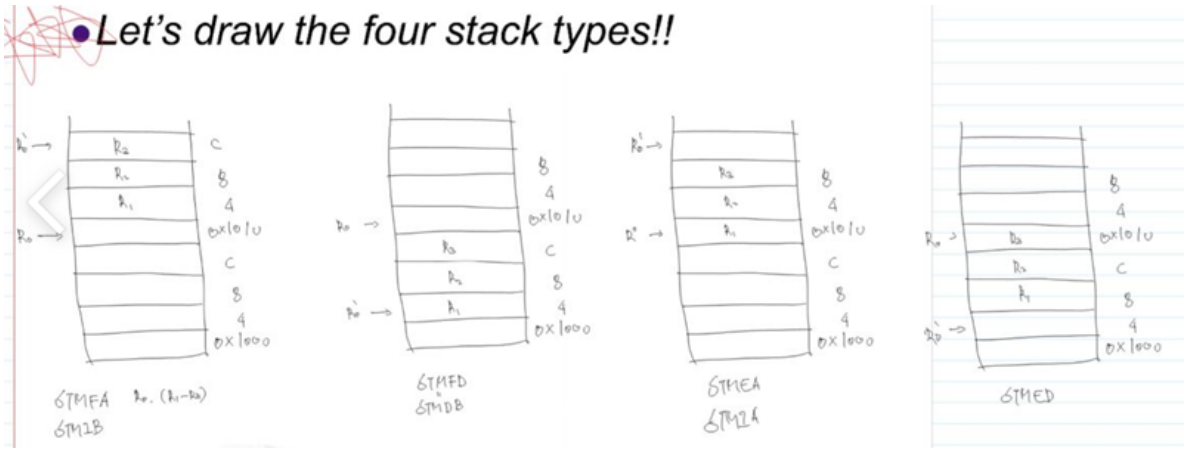
스택이 위로 올라감: ascending

스택이 아래로 내려감 : descending

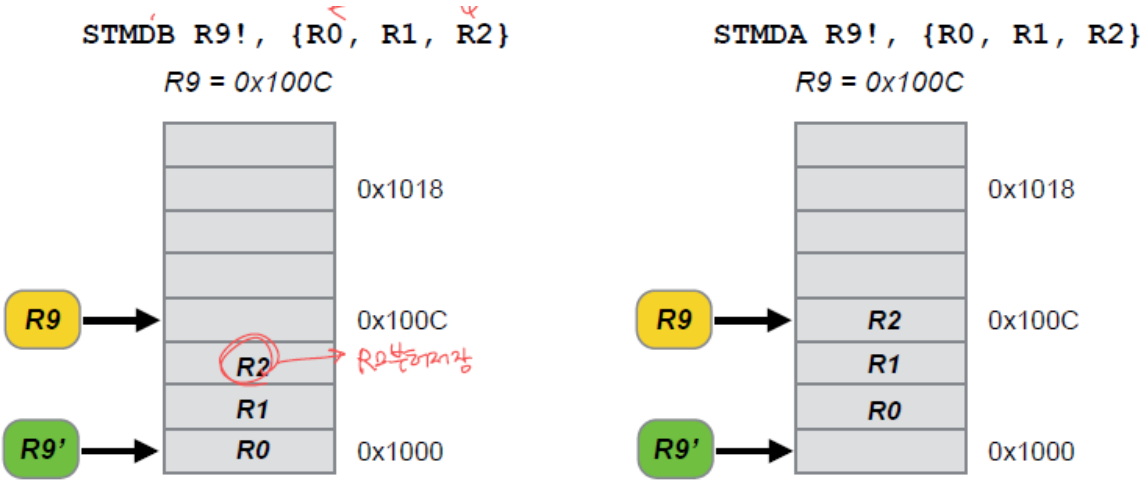
LDM / STM Examples



1. STMIB R9 !, {R0, R1, R2} : `데이터 저장 전 address 증가`
    1. r9 -> 0x1010 증가 후 r0 저장
    2. r9 -> 0x1014 증가 후 r1저장
    3. r9 -> 0x10108 증가 후 r2 저장
    4. {!}, auto-update 옵션이 있으면 R9 값을 0x1018로 변경 (! 없으면 변함 없음)
2. STMIA R9 !, {R0, R1, R2} : `데이터 저장 후 address 증가`
    1. r0 저장 후, r9 -> 0x1010 증가
    2. r1 저장 후, r9 -> 0x1014 증가
    3. r2 저장 후, r9 -> 0x1018 증가
    - {!}, auto-update 옵션이 있으면 R9 값을 0x1018로 변경



LDM / STM Examples

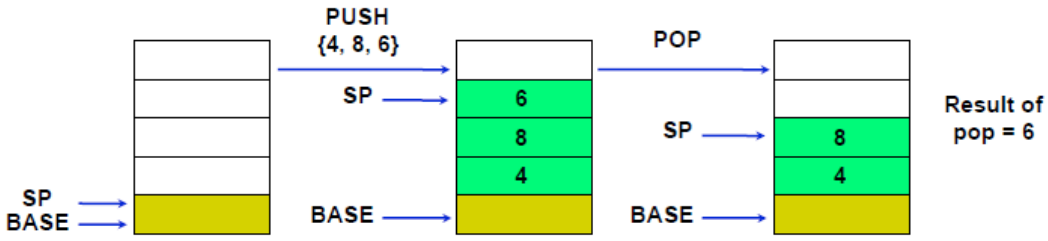


- Descending은 맨 뒤 부터 저장
- STMDB R9 !, {R0, R1, R2} : `address <register_list> 개수 만큼 감소해 놓고, address를 증가 하면서 데이터 저장`
  1. address를 0x1000로 감소
  2. 0x1000에 r0 저장 후, address 증가
  3. 0x1004에 r1 저장 후, address 증가
  4. 0x1008에 r2 저장 후, address 증가
  - {!}, auto-update 옵션이 있으면 R9 값을 0x1000로 변경
- STMDA R9 !, {R0, R1, R2} : `address <register_list> 개수 만큼 감소해 놓고, address를 증가하면서 데이터 저장`
  1. address를 0x1000으로 감소
  2. address 증가 후, 0x1004에 r0 저장
  3. address 증가 후, 0x1008에 r1 저장
  4. address 증가 후, 0x100C에 r2 저장

  {!}, auto-update 옵션이 있으면 R9 값을 0x1000로 변경

Stack Operations

- Push and Pop
  - 새로운 데이터를 "Push"를 통해 "top"위치에 삽입하고, "POP"을 통해 가장 최근에 삽입된 데이터를 꺼내는 자료구조 형태
    - Base pointer : 스택의 bottom 위치를 지정
    - Stack pointer : 스택의 top 위치를 지정



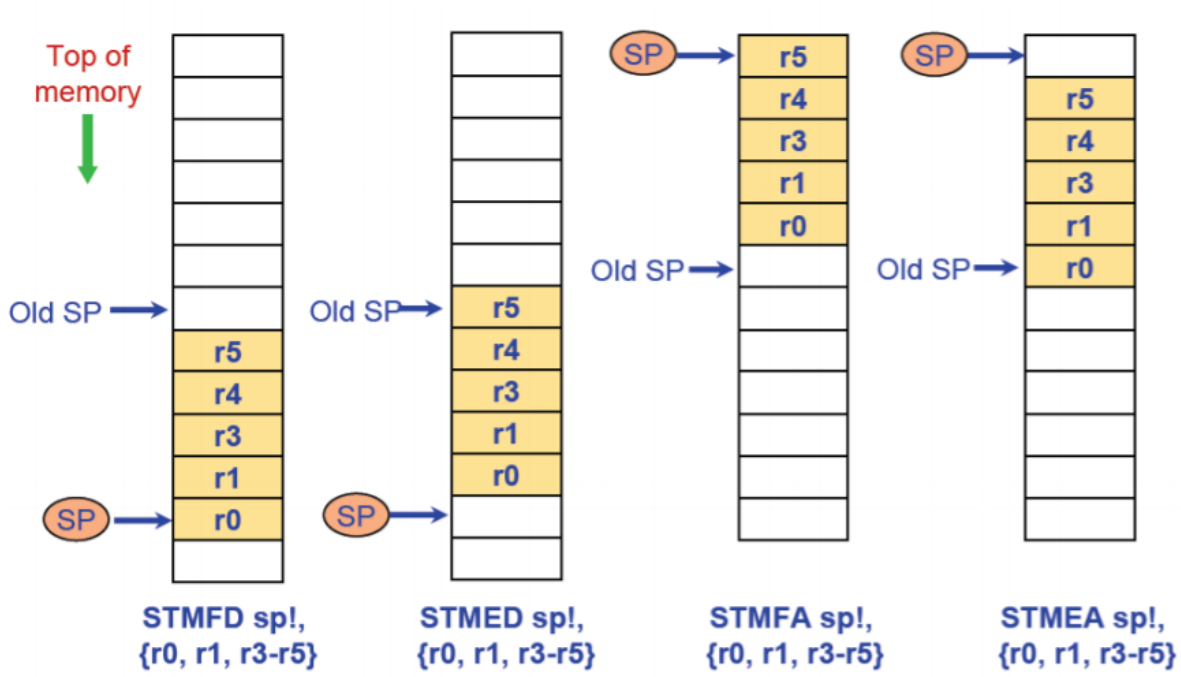
Stack Types

- Full stack : stack pointer가 stack의 마지막 주소를 지정

  stack pointer가 stack의 top을 지정하고 push가 필요하면 address 증가/감소하면서 사용
  - stack pointer의 위치가 마지막 데이터값으로 채워진 곳을 가리킴

    - 다음 push를 할때 먼저 stack pointer가 decrease/increase해야함
    - FD(Fully Descending) AD(Fully Ascending) stack이 있음
- Empty stack : stack pointer가 stack의 다음 주소를 지정
    - stack pointer의 위치가 비어있어 다음 push로 저장되는 data가 저장될 것임
    - push 작동 후 stack pointer가 increased/decreased 되어 다른 empty 위치를 가리킴
    - ED(Empty Descending), EA(Empty Ascending) stack이 있음

Stack Types



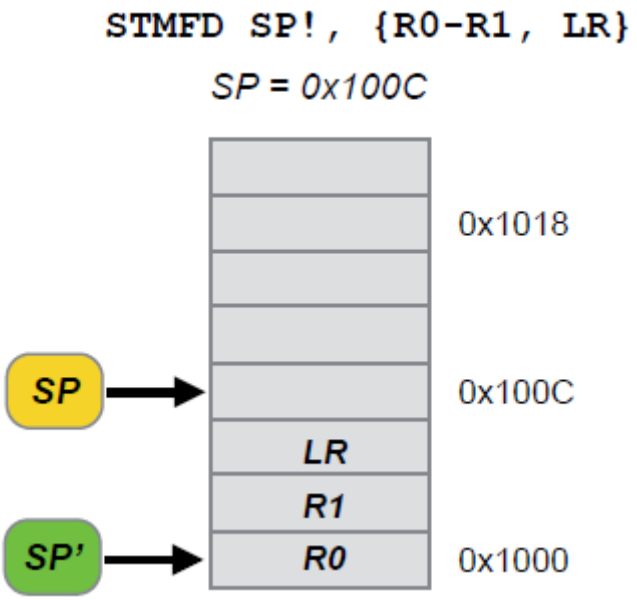
Stack Operation Examples

fuction call 을 했을때 이렇게 한 줄의 명령어로 깔끔하게 들어갈 수 있습니다.

어셈블러에서 LDM과 STM을 활용하고있다 : LR에 기록됐던 return address가 pc에 자동으로 세팅돼서 context를 resotre 하면서 그 즉시 원래 위치로 jump 하는 것을 한 명령어에서 표현하고 있습니다.

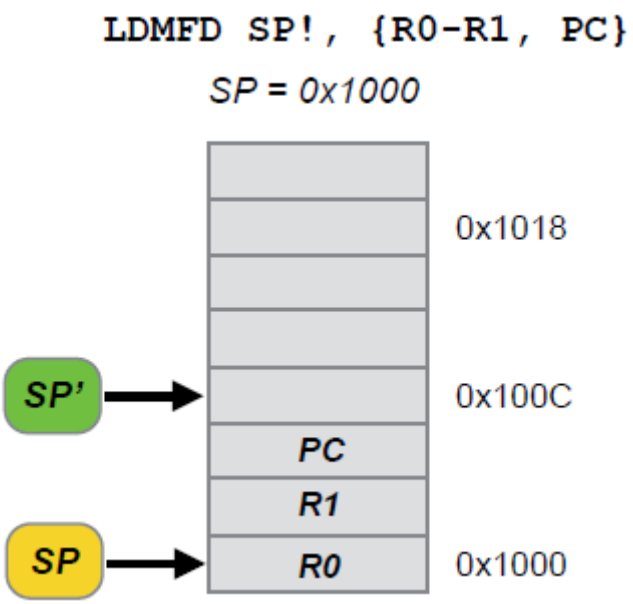
내가 돌아갈 주소를 LR에 누군가가 먼저 해놧을겁니다.(가정)

- STM은 reg 표현과 stack 표현이 연동됨.
- reg 표현에서 STM과 LDM은 상호 반대임
- stack 표현에서 STM과 LDM은 상호 같음.
- Stack의 Push 동작



- stack에 context 정보 저장
  1. address를 0x1000로 감소
  2. 레지스터 값 저장 후, address 증가
  3. 링크 레지스터 저장 후 address 증가
  4. Stack의 위치를 0x1000로 변경

- Stack의 Pop 동작



- stack에서 context 정보를 읽음
  1. 레지스터 값을 읽은 후 address 증가
  2. 링크 레지스터 (LR)값을 읽어 PC에 저장
  3. Stack의 위치를 0x100C로 변경

More Examples

LDM example은 별거 없습니다. 그냥 들어가는거 확인하면 되고요 - 진짜 별거없음

```
PRE     mem32[0x80018] = 0x03
        mem32[0x80014] = 0x02
        mem32[0x80010] = 0x01
        r0 = 0x00080010
        r1 = 0x000000C0
        r2 = 0x000000C0
        r3 = 0x000000C0

LDMIA r0!, {r1-r3}

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

More Examples

아트다. 예술적이다

```
loop
LDMIA r9 !, (r0-r7)     ; 메모리로부터 r0~r7의 값을 읽음
                        ; r0~r7에 세팅
STMIA r10 !, (r0-r7)
CMP r9, r11             ; src의 마지막 부분까지 도착했는지 보는거에요 r9: 동적, r11 끝부분
BNE loop                ; r9와 r11이 같지 않다면 다시 loop
                        ; 딱 맞아 떨어지지 않는다면 폭주해버리는 위험성
```