

Embedded Systems Design

Lecture 8

Yongsoo Joo

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

- 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	

- Conditional executions are possible
 - Ex) LDREQ

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
- **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

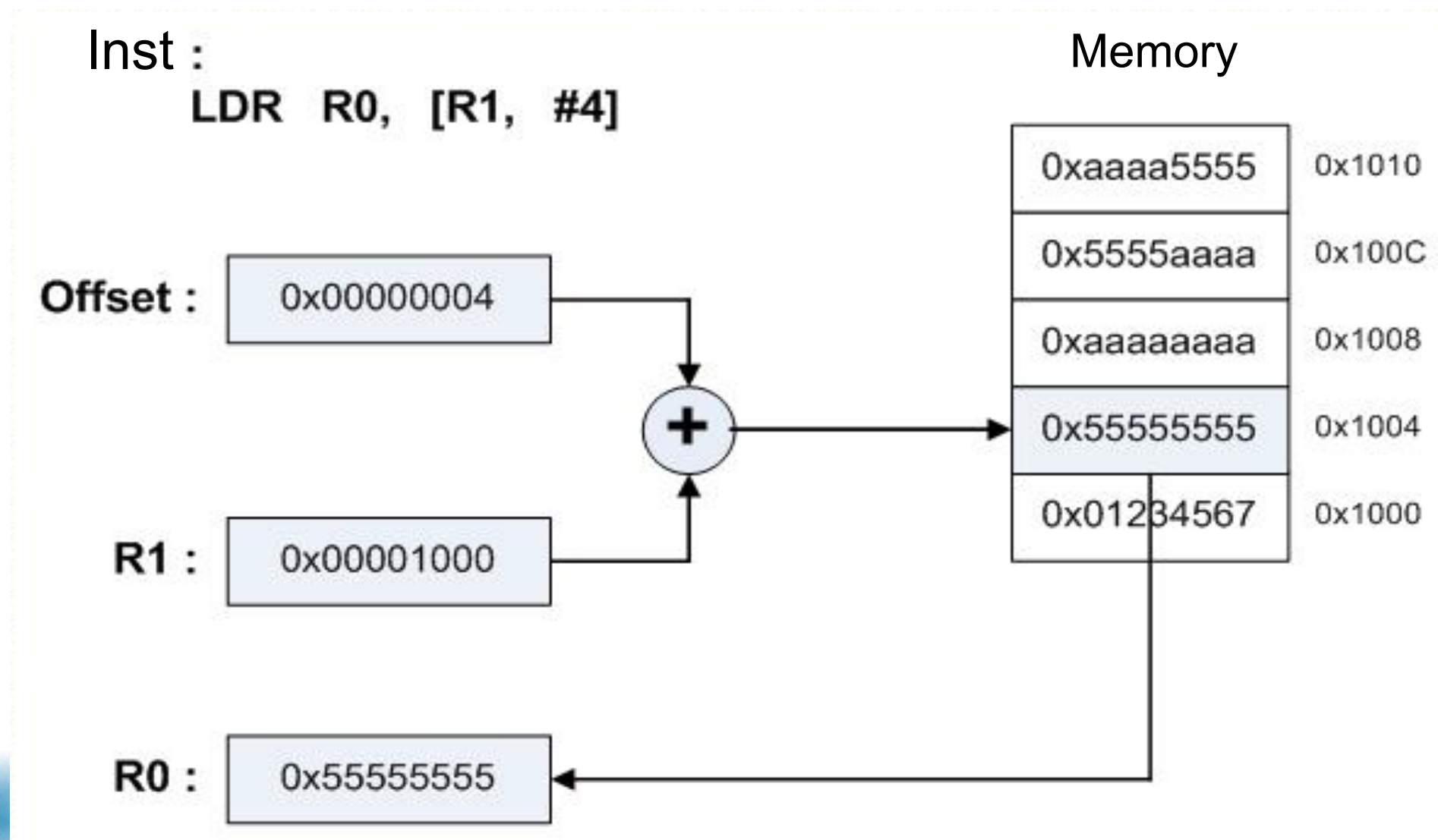
- The target address is calculated first before actual accessing the address
 - $[Rn, \text{Offset}] \{!\}$
 - '!' means auto-update

- Post-Indexed

- The target address is calculated/modified after the data access for load/store has been performed
 - $[Rn], \text{Offset}$

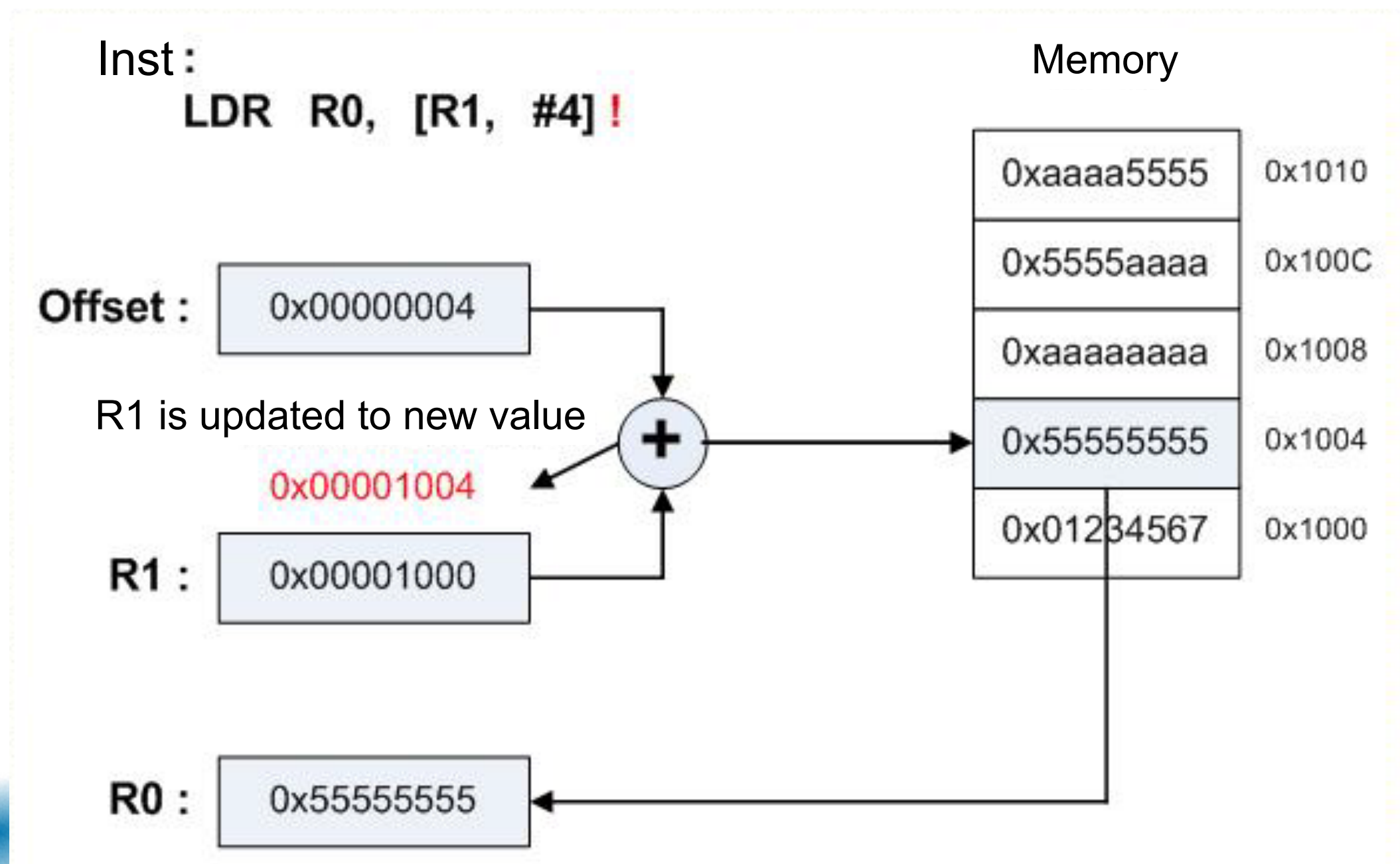
Pre-Indexed Addressing Mode

- The target address is calculated first and then data access is performed
 - The value of base register is not changed after data access if auto-update is not set



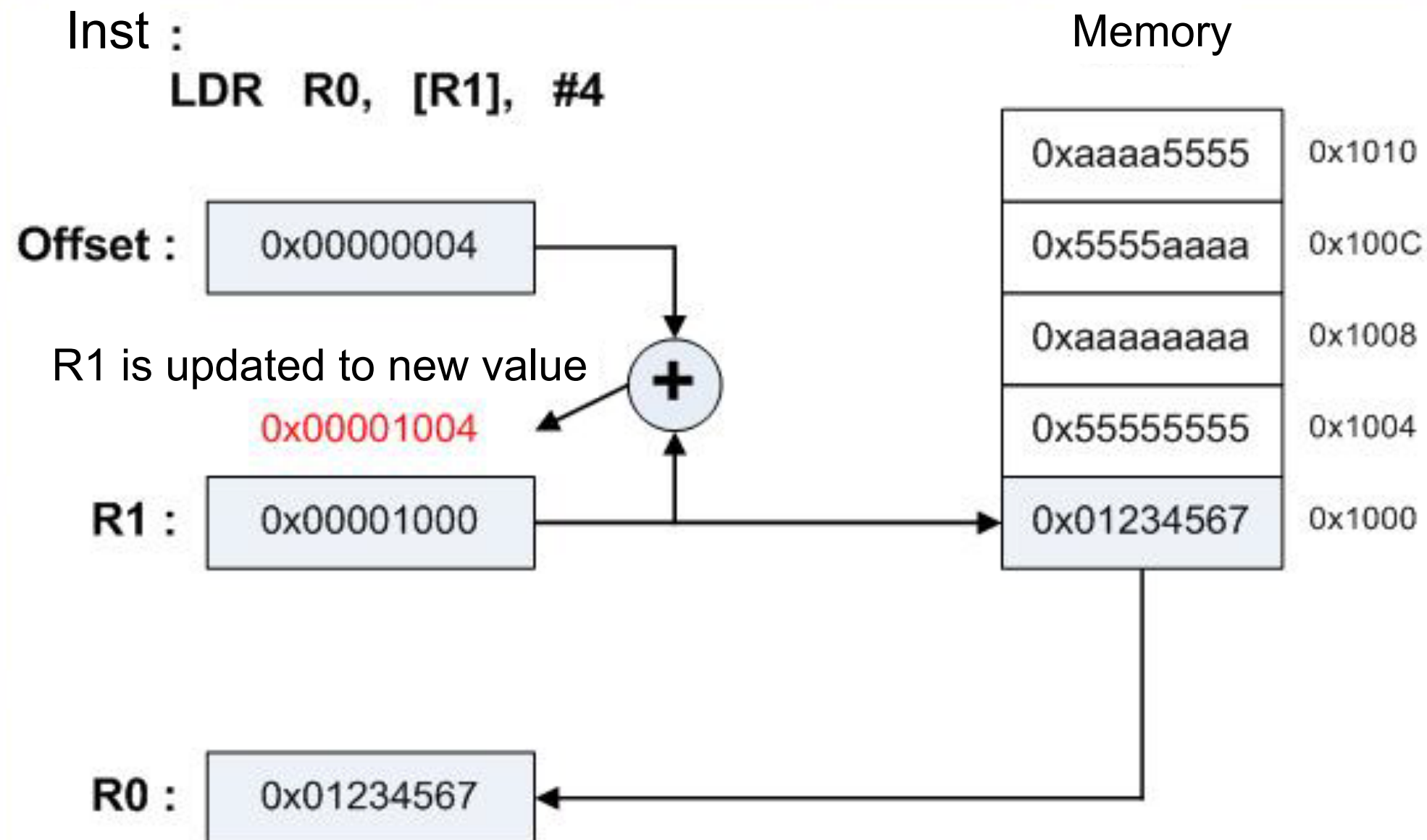
Pre-Indexed with Auto-Update

- Same as pre-indexed except the update of base register after data access



Post-Indexed Addressing Mode

- Data access to base register is performed first and then the value of base register is updated to new value with offset is added



PC-relative Addressing Mode

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

```
LDR r1, label
```

```
.....
```

```
Label:
```

```
DCD 0x12345678
```



```
LDR r1, [PC, #offset]
```

```
.....
```

```
DCD 0x12345678
```

- 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

```
LDR r1, =0x12345678
```



```
LDR r1, [PC, #offset]
```

```
.....
```

```
DCD 0x12345678
```

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

- 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>
- 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

- 
actual operation

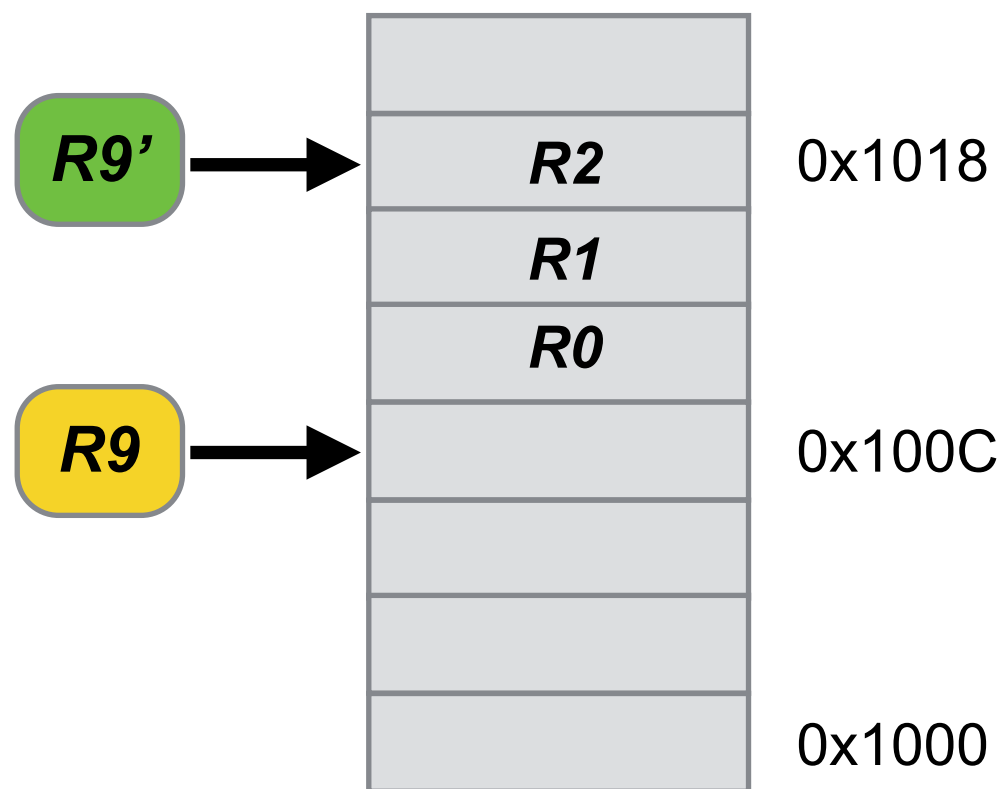
Addressing Modes in LDM/STM

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

LDM/STM Examples

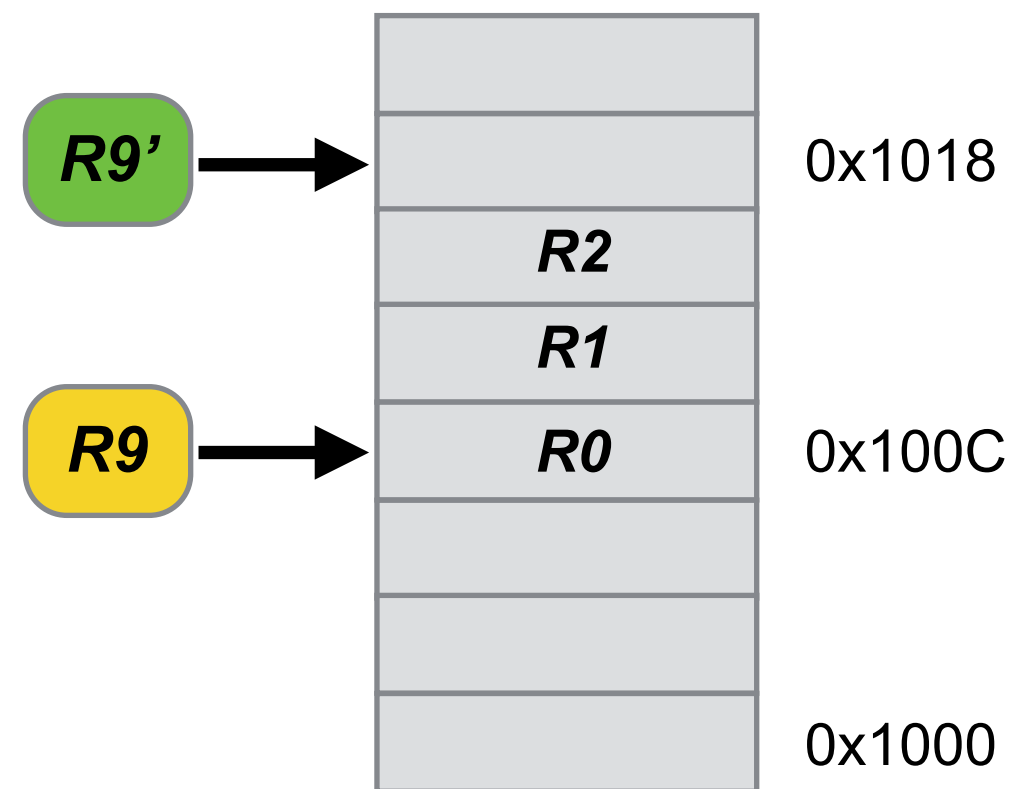
STMIB R9!, {R0, R1, R2}

R9 = 0x100C



STMIA R9!, {R0, R1, R2}

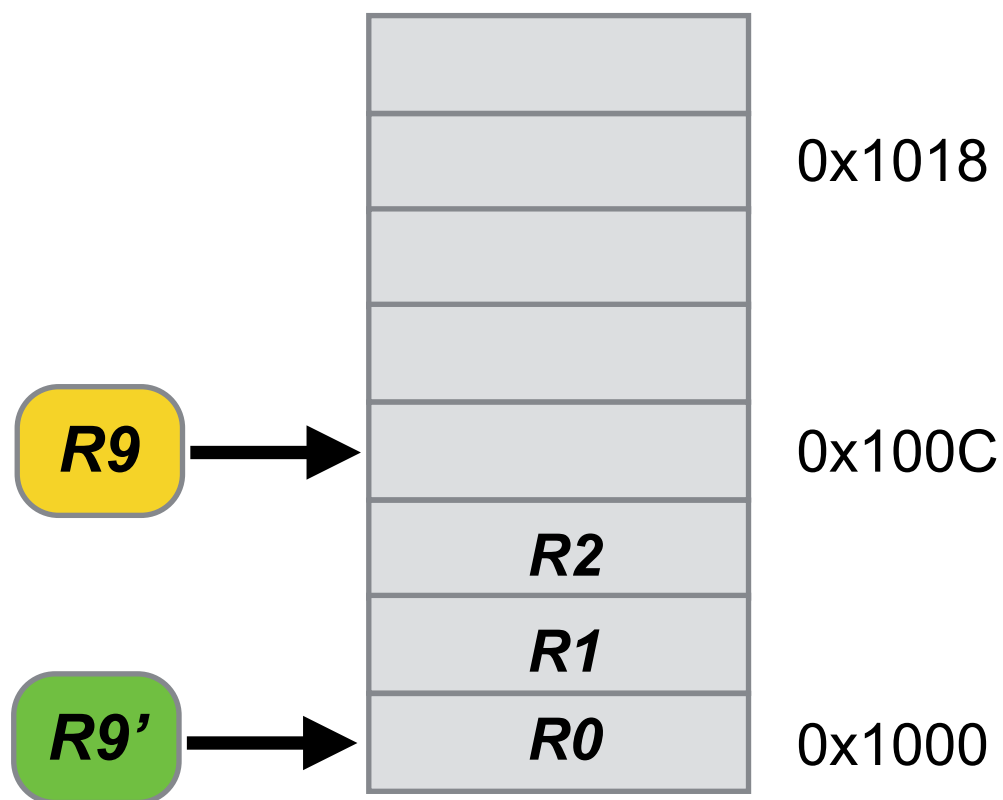
R9 = 0x100C



LDM/STM Examples

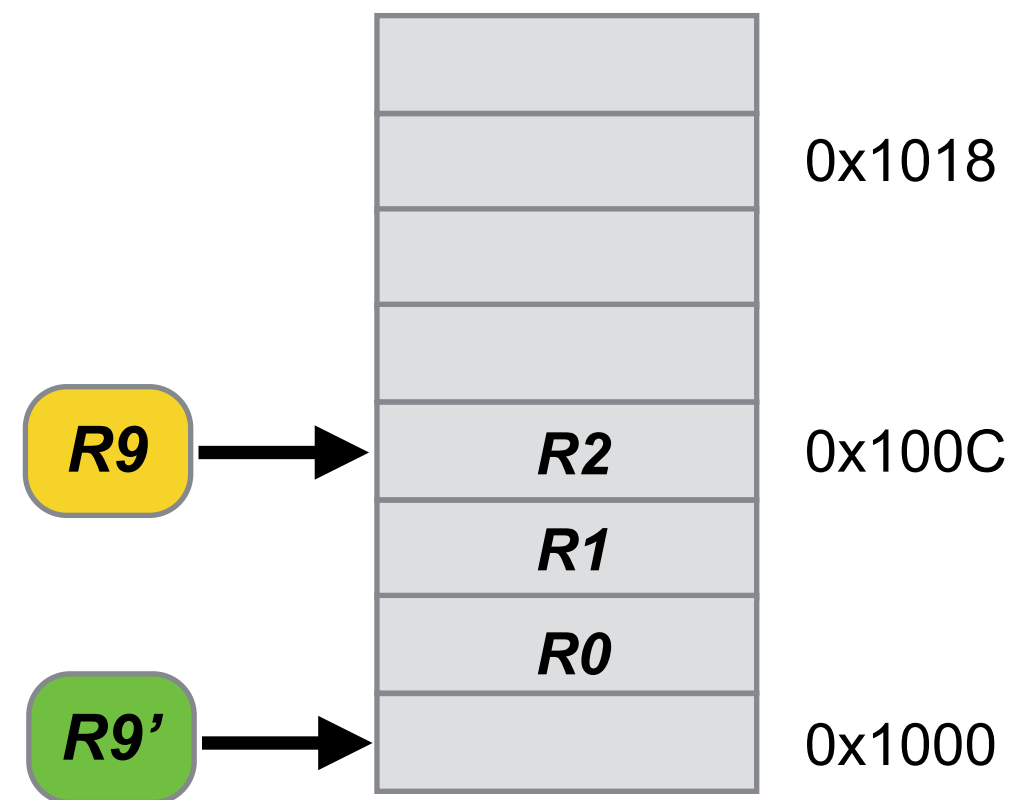
STMDB R9!, {R0, R1, R2}

R9 = 0x100C



STMDA R9!, {R0, R1, R2}

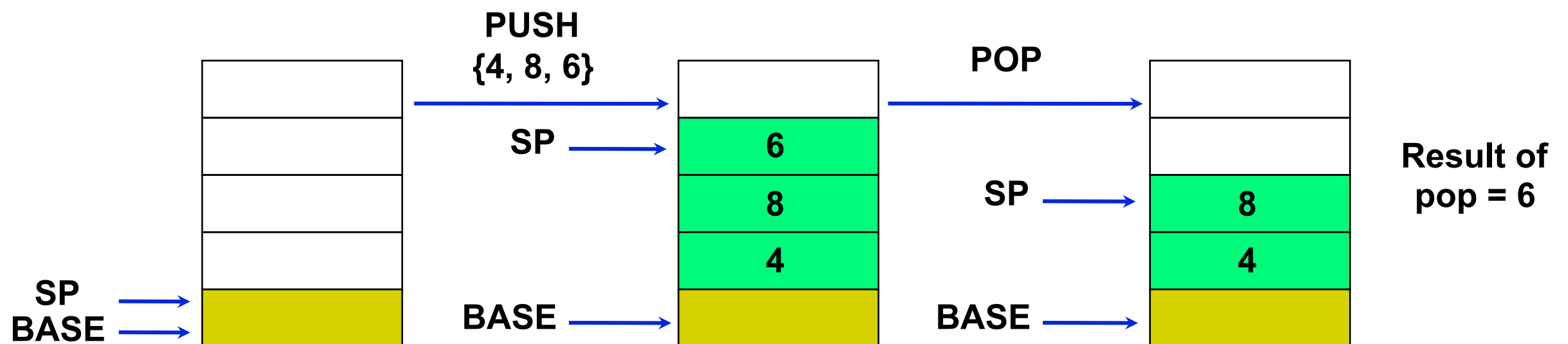
R9 = 0x100C



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
 - Full Descending and Full Ascending stacks (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
 - Empty Descending and Empty Ascending stacks (ED/EA stacks)

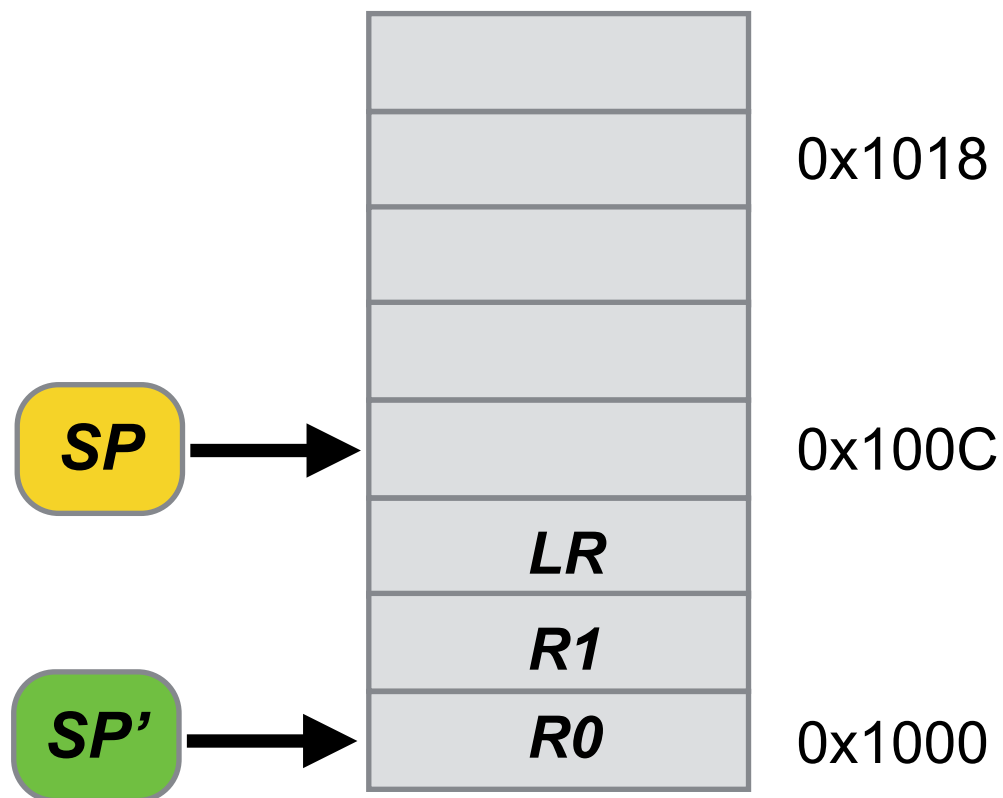
Stack Types

- *Let's draw the four stack types!!*

Stack Operation Examples

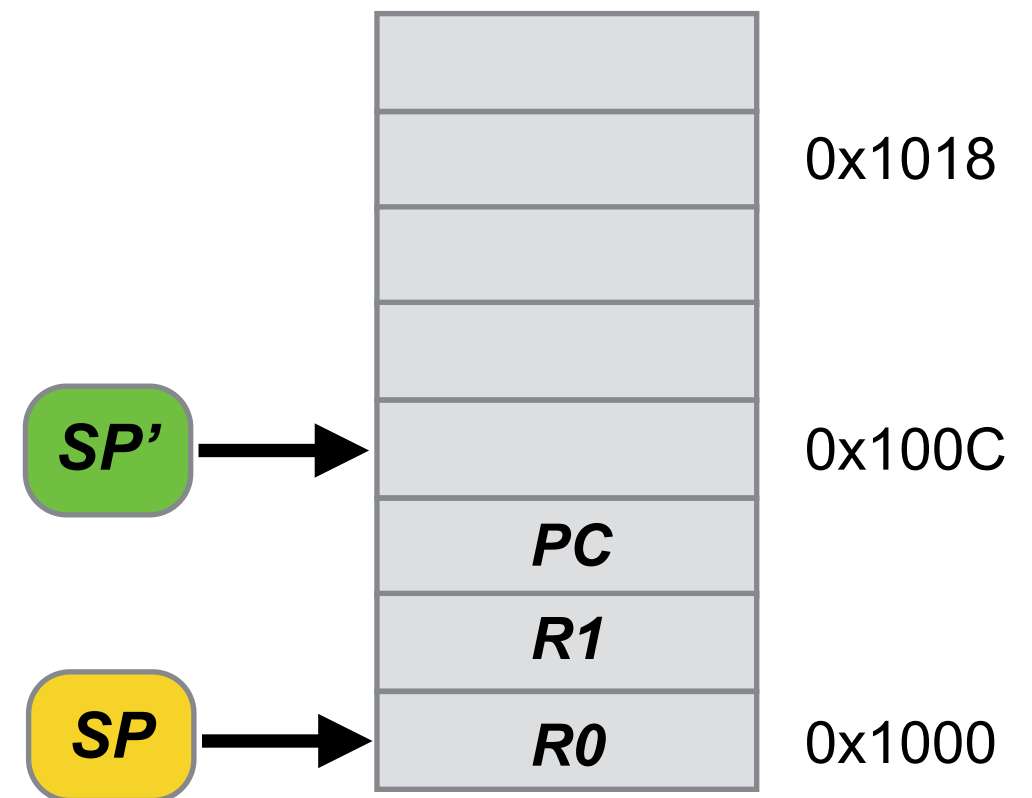
STMFD SP!, {R0-R1, LR}

SP = 0x100C



LDMFD SP!, {R0-R1, PC}

SP = 0x1000



More Examples

- LDM example

```
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
```


More Examples

- Memory copy

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

