

Data Transfer (4A)

Copyright (c) 2014 - 2020 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Based on

Introduction to ARM Cortex-M Microcontrollers – Embedded Systems,
J. W Valvano

Memory Objects

Memory object type	Register	Example operand
Constants in <u>code</u> space	PC	=Constant [PC, #28]
Local variables on the <u>stack</u>	SP	[SP, #0x04]
Global variables in RAM	R0-R12	[R0]
I/O ports	R0-R12	[R0]

Address loading pseudo-instructions

```
ADR {cond} Rd, label      (address)
ADRL {cond} Rd, expression (address arithmetic)
LDR {cond} Rd, =label     (=address)
LDR {cond} Rd, =number    (=constant)
```

<https://stackoverflow.com/questions/42065155/how-to-replace-ldr-with-adr-in-assembler>

ADR Rd, label

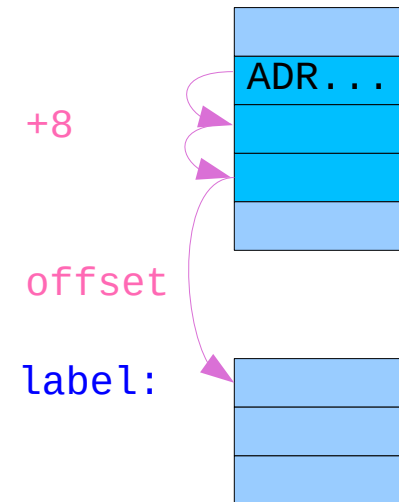
ADR {cond} Rd, label

ADR is to get the **address** of the **literal pool** (type constant) to a register.

The **literal pool** in code area is typically after the end of **functions**

ADR Rd, label can be translated into

➔ **ADD Rd, pc, #offset**



<https://stackoverflow.com/questions/42065155/how-to-replace-ldr-with-adr-in-assembly>

ADRL Rd, expression

ADRL {cond} Rd, expression

The assembler converts an **ADRL Rd, label** into

two data processing instructions that load the address, if it is in range

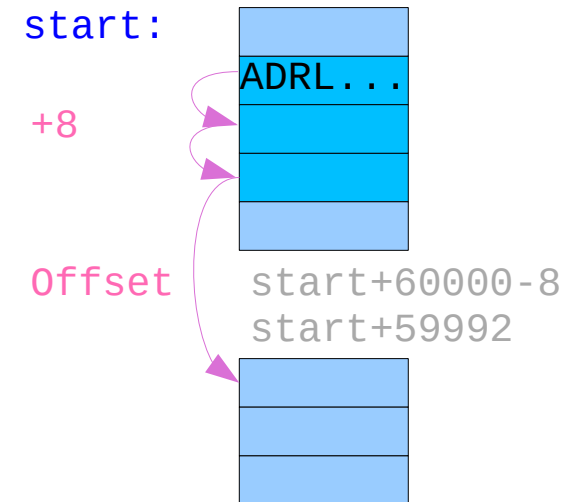
```
start MOV    r0,#10
      ADRL   r4,start + 60000
```



```
; ADD r4,pc,#0xe800
; ADD r4,r4,#0x254
```

$60000 - 8 - 4 = 59988 = \&ea54$
 $\&e800 + \&254 = \&ea54$

prefix **&** = prefix **0x**



<https://stackoverflow.com/questions/42065155/how-to-replace-ldr-with-adr-in-assembler>

LDR Rd, =label

```
LDR {cond} Rd, =label      (=address)
```

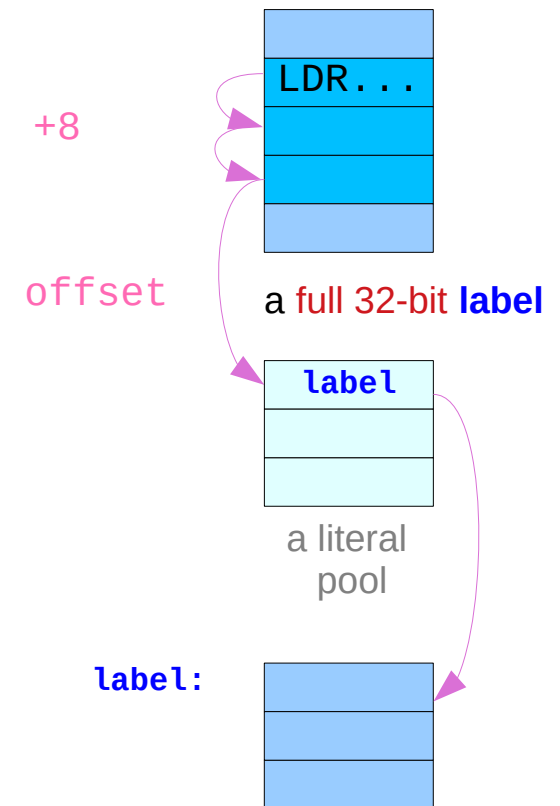
placing the address **label** in a literal pool

```
LDR Rd, =label
```

→

```
LDR Rd, [pc, #offset]
```

the offset to a **literal pool**



<https://stackoverflow.com/questions/42065155/how-to-replace-ldr-with-adr-in-assembly>

LDR Rd, =number

```
LDR {cond} Rd, =number      (=constant)
```

placing the **number** in a literal pool

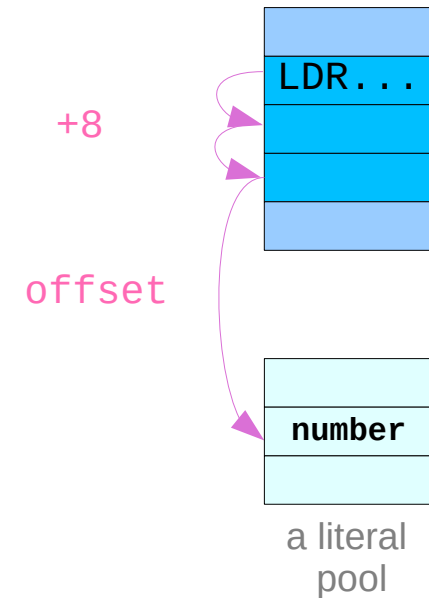
```
LDR Rd, =number
```

```
LDR Rd, [pc, #offset]
```

the offset to a **literal pool**

```
MOV Rd, #imm16
```

for a small range number,



<https://stackoverflow.com/questions/42065155/how-to-replace-ldr-with-adr-in-assembler>

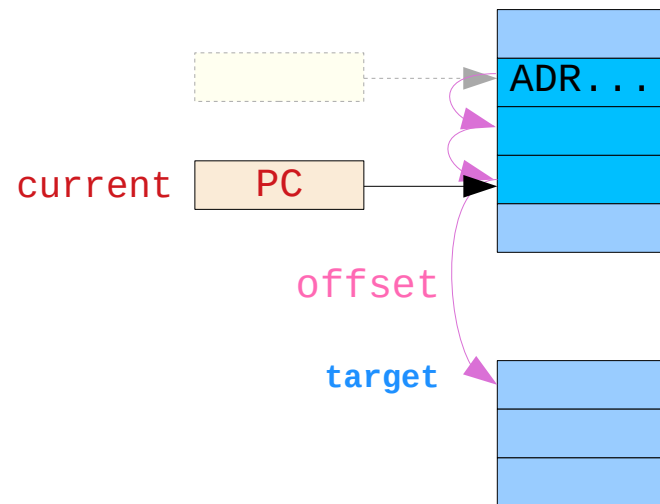
PC-relative address

ADR / LDR instruction
adds or subtracts an **offset**
to the current **PC** value
to form a **PC-relative address**

at the time of executing
a current instruction,
PC has been advanced
by **2** instructions forward (+8)

$$\text{pc} - \text{target} = \text{offset}$$

In ARM state, the value of the PC
is the address of the current
instruction **plus 8** bytes.



<https://stackoverflow.com/questions/15774581/getting-an-label-address-to-a-register-on-arm>

Getting a label address into a register

```
target:  
    .long 0xfeadbeef  
  
adr    r0, target  
adr l  r0, target  
ldr    r0, =target  
sub    r0, pc, #(.+8-target)
```

- 1) and 2) are very similar and generate `sub r0, pc, #target`.
- 3) puts a `long` in a `literal pool` and loads this via `ldr r0, [pc, #offset2]` or it may use a `mov r0, #offset2` if the assembler finds it can (usually an aligned label, like at 0x8000).
- 4) is to manually calculated a `full 32-bit absolute address`

<https://stackoverflow.com/questions/15774581/getting-an-label-address-to-a-register-on-arm>

ADR / ADRL offsets

target:

```
.long 0xf0000000
```

```
adr    r0, target
```

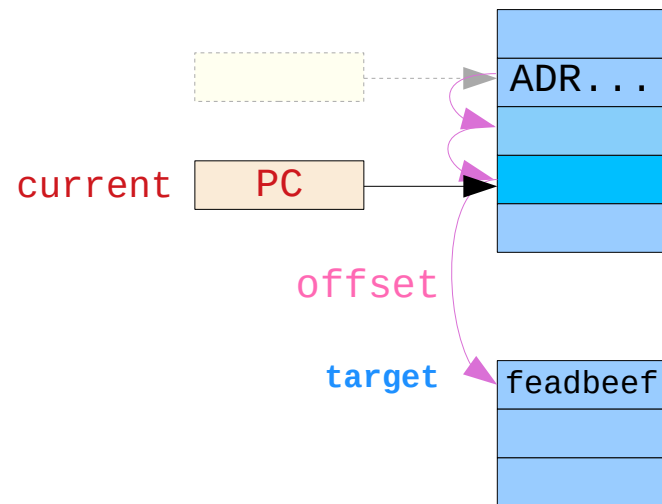
```
adr.l  r0, target
```

1) and 2) are very similar and generate **sub r0, pc, offset**

$$r0 \leftarrow pc - target$$
$$r0 \leftarrow offset$$

pc - target = offset

In ARM state, the value of the PC is the address of the current instruction **plus 8 bytes**.



<https://stackoverflow.com/questions/15774581/getting-an-label-address-to-a-register-on-arm>

ADR vs ADRL

The difference between **adr** and **adr l** comes from **immediate operands**.

immediate operands are **8bits** rotated by a multiple of two.

So if the address is **far**, you may need to perform two instructions (**adr l**)

adr l will usually be faster than the **ldr** variant (**ldr =target**) which get a **full 32-bits** address in the **literal pool**

<https://stackoverflow.com/questions/15774581/getting-an-label-address-to-a-register-on-arm>

LDR offset

target:

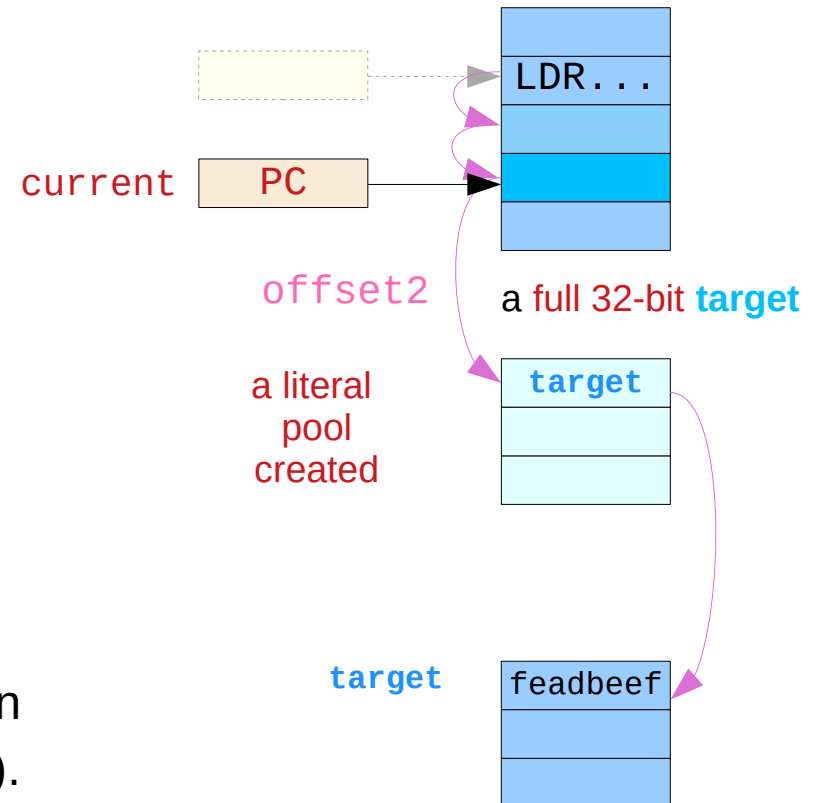
```
.long 0xfeadbeef
```

```
ldr    r0, =target
```

puts a **long** in a **literal pool**
to hold a full 32-bit address of **target**
and loads this address via

```
ldr r0, [pc, #offset2]
```

Or it may use a **MOV** if the assembler finds it can
(usually an aligned label, like at 0x8000).



<https://stackoverflow.com/questions/15774581/getting-an-label-address-to-a-register-on-arm>

Manual computing an address

target:

```
.long 0xfeadbeef
```

```
sub    r0, pc, #(.+8 - target)
```

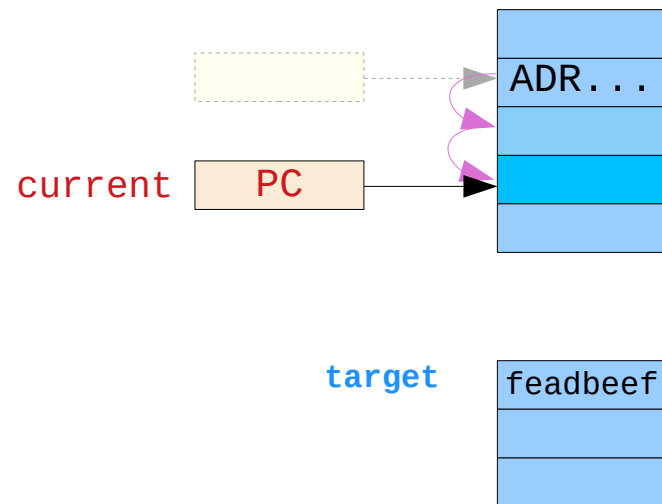
`.` : address of the current instruction
address of the sub ...

`+.8` : where the current PC points to

$$PC - (+8 - target) =$$
$$PC - (PC - target) =$$

a full 32-bit target address

In ARM state, the value of the PC is the address of the current instruction **plus 8** bytes.



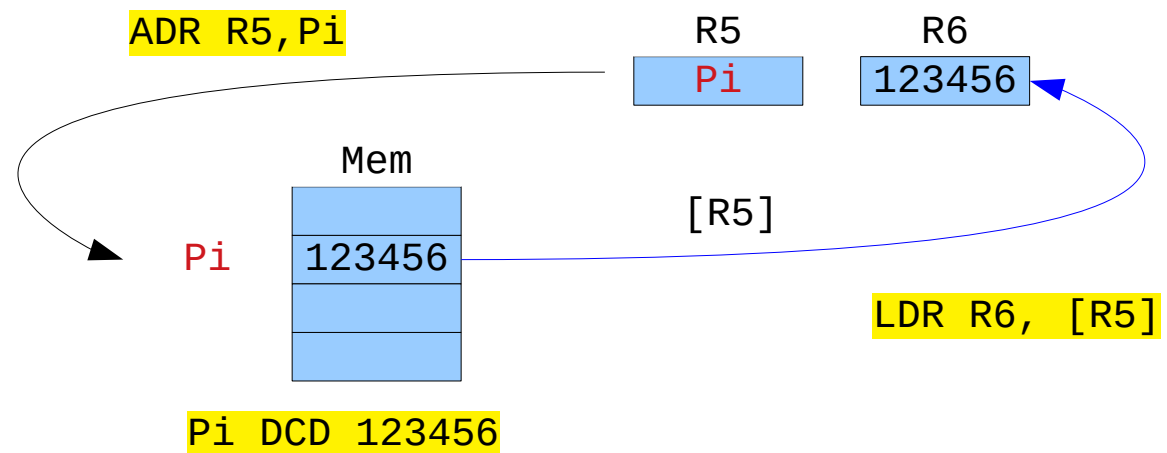
<https://stackoverflow.com/questions/15774581/getting-an-label-address-to-a-register-on-arm>

ADR Label Example

```
Access  ADR R5, Pi ; R5 points to Pi
        LDR R6, [R5] ; R6 = 123456
        ...
        BX LR
Pi      DCD 123456 ; literal pool
```

Pi the address of the location where the value N is stored

R5 ← PC-relative Pi



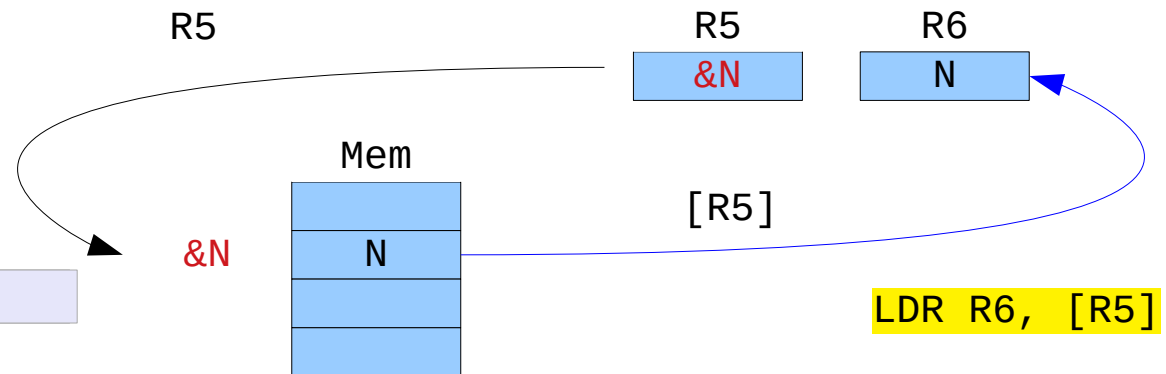
LDR =number example

In `LDR R5, =0x12345678`
`LDR R6, [R5]`

`&N` the address of the location where the value N is stored

...
`BX LR`

In `LDR R5, [PC, #16]`
`LDR R6, [R5]`



...
`BX LR`
`DCD 0x12345678`

`R5 ← PC-relative &N`

Data Transfer Types

B	Unsigned 8-bit byte
SB	Signed 8-bit byte
H	Unsigned 16-bit halfword
SH	Signed 16-bit halfword
D	64-bit data

Data Transfer Examples

LDR{type}{cond} Rd, [Rn]	[Rn]
STR{type}{cond} Rd, [Rn]	[Rn]
LDR{type}{cond} Rd, [Rn, #n]	[Rn + #n]
STR{type}{cond} Rt, [Rn, #n]	[Rn + #n]
LDR{type}{cond} Rd, [Rn, Rm, LSL #n]	[R+(Rm<<#n)]
STR{type}{cond} Rd, [Rn, Rm, LSL #n]	[R+(Rm<<#n)]

{type} = {B|SB|H|SH|D}

Data Transfer Examples

```
MOV{S}{cond}    Rd,    <op2>
MOV    {cond}    Rd,    #im16
MVN{S}          Rd,    <op2>
```

If **S** is specified, the condition code flags are updated on the result of the operation

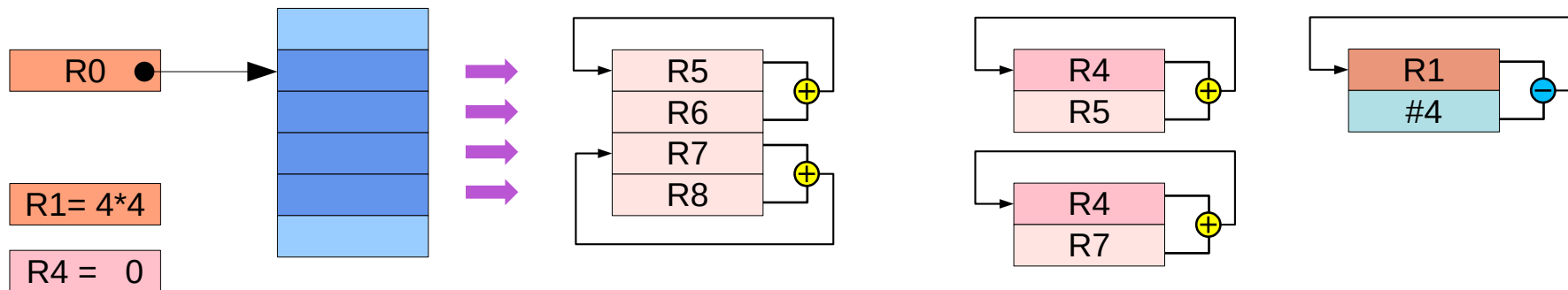
S cannot be used with 16-bit immediate operand

These belong to data processing instructions

LDM / STM examples (1)

; R0 holds address of first integer in array
; R1 holds array's length;
; fragment works only if length is multiple of 4

```
addInts      MOV      R4, #0
addLoop      LDMIA   R0!, { R5-R8 }
             ADD     R5, R5, R6
             ADD     R7, R7, R8
             ADD     R4, R4, R5
             ADD     R4, R4, R7
             SUBS   R1, R1, #4
             BNE    addLoop
```



<http://www.cburch.com/books/arm/>

LDM / STM examples (2)

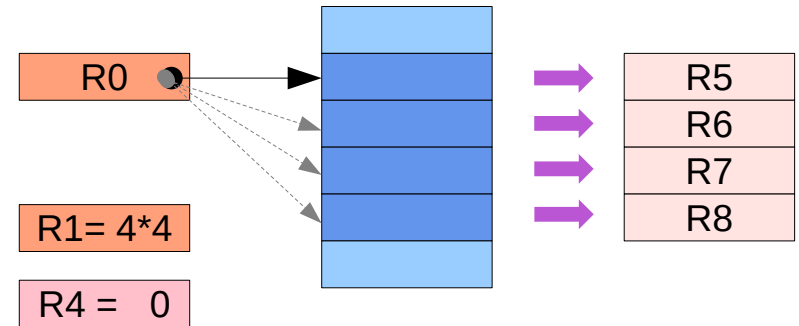
the ARM processor looks into the **R0** register for an **address**.

It loads into **R5** the four bytes starting at that address,
into **R6** the next four bytes,
into **R7** the next four bytes,
and into **R8** the next four bytes.

```
R5 := mem32[R0 + 0]
R6 := mem32[R0 + 4]
R7 := mem32[R0 + 8]
R8 := mem32[R0 + 12]
```

Meanwhile, **R0** is stepped forward by 16 bytes,
so with the next iteration the **LDMIA** instruction
will load the next four words into the registers.

```
R0 := R0 + 12
```



```
LDMIA R0!, { R5-R8 }
```

LDM / STM examples (3)

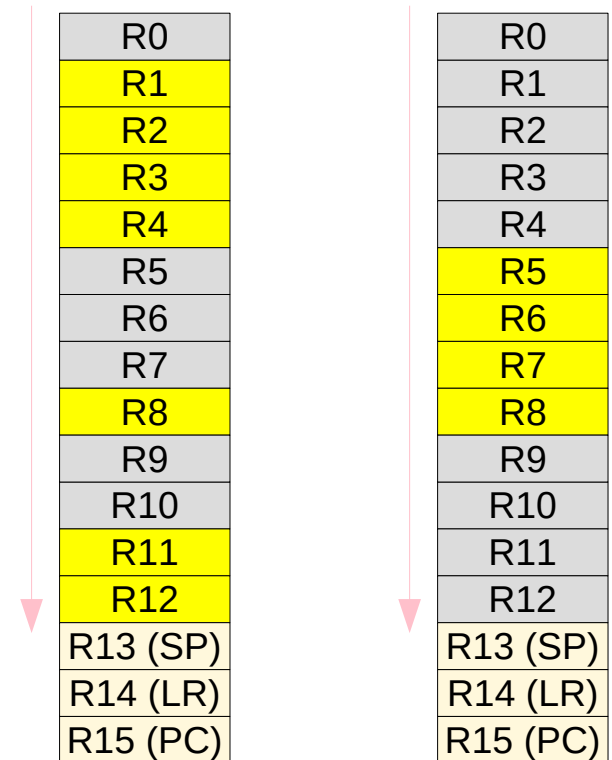

Inside the braces `{ }` can be any list of registers, using **dashes** to indicate ranges of registers, and using **commas** to separate ranges.

The **order** in which the registers are listed is not significant;

Thus, the instruction **LDMIA R0!, { R1-R4, R8, R11-R12 }** will load seven words from memory.

even if we write **LDMIA R0!, { R11-R12, R8, R1-R4 }**, **R1** will receive the first word loaded from memory.

{ R1, R2, R3, R4, R8, R11, R12 }



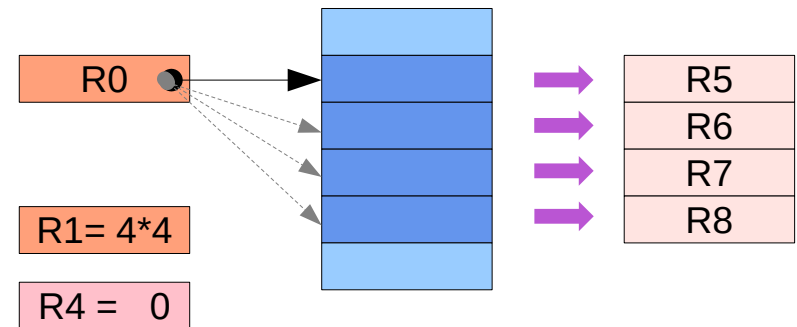
LDMIA R0!, { R5-R8 }

<http://www.cburch.com/books/arm/>

LDM / STM examples (4)

If the **exclamation** mark **!** following **R0** is omitted, then the address register **R0** is not altered. **R0** would continue pointing to the first integer in the array.

we want **R0** to change so that it is pointing to the next four integers for the next iteration, the exclamation point should be included



LDMIA R0!, { R5-R8 }

*equivalent instructions without !
pre-indexed*

```
LDR    R5, =[R0, #0]
LDR    R6, =[R0, #4]
LDR    R7, =[R0, #8]
LDR    R8, =[R0, #12]
ADD    R0, R0, #12
```

*equivalent instructions with !
post-indexed*

```
LDR    R5, =[R0], #4
LDR    R6, =[R0], #4
LDR    R7, =[R0], #4
LDR    R8, =[R0], #4
```

<http://www.cburch.com/books/arm/>

LDM / STM examples (5)

STMIA stores several registers into memory.

In the following example, every number in an array is shifted into the next element;

the array <2,3,5,7> becomes <0,2,3,5>.

<http://www.cburch.com/books/arm/>

LDM / STM examples (6)

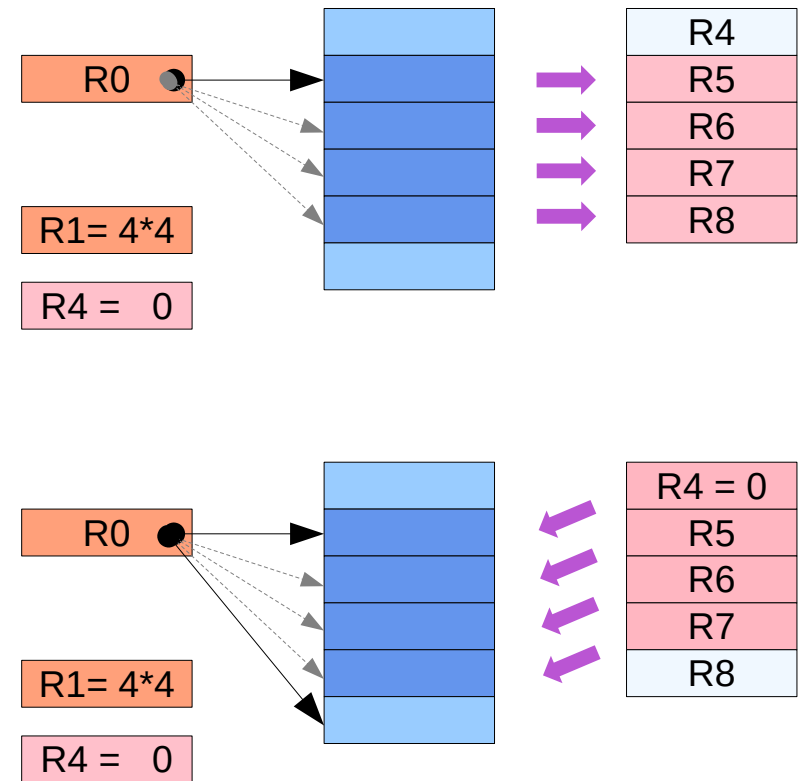
; R0 holds address of first integer in array
; R1 holds array's length;
; fragment works only if length is multiple of 4

```
shift    MOV      R4, #0
shLoop   LDMIA   R0, { R5-R8 }
          STMIA   R0!, { R4-R7 }
          MOV     R4, R8
          SUBS   R1, R1, #4
          BNE    shLoop
```

Notice how the **LDMIA** instruction does not have !
so that **R0** isn't modified.

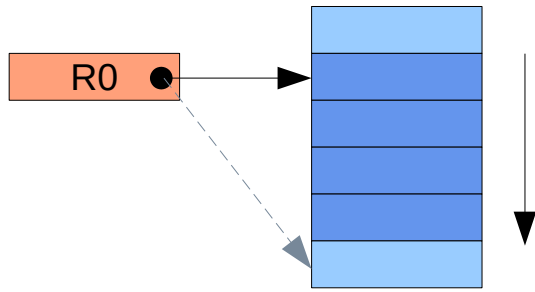
Thus, the **STMIA** stores into the same range of addresses
that were just loaded into the registers.

The **STMIA** instruction has ! because **R0** must be modified
for the next iteration of the loop.

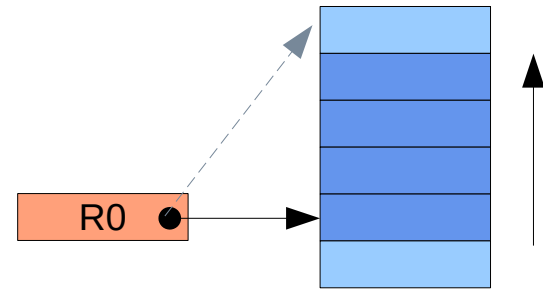


<http://www.cburch.com/books/arm/>

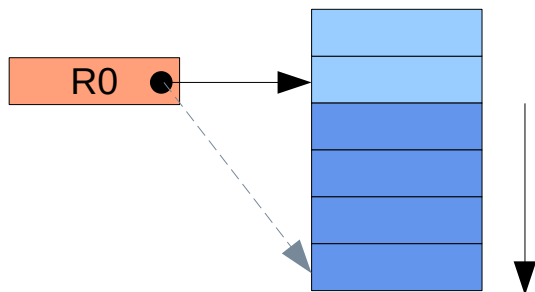
LDM / STM examples (7)



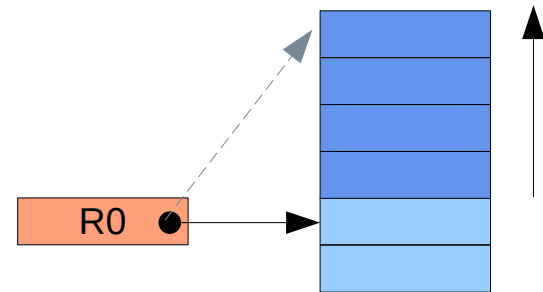
LDMIA, STMIA
Increment **after**



LDMDA, STMDA
Decrement **after**



LDMIB, STMIB
Increment **before**



LDMDB, STMDB
Decrement **before**

<http://www.cburch.com/books/arm/>

LDM / STM examples (8)

LDMIA, STMIA Increment after

loading from the named address and
storing into *increasing* addresses.

LDMIB, STMIB Increment before

loading from *four more* than the named address and
storing into *increasing* addresses.

LDMDA, STMDA Decrement after

loading from the named address and
storing into *decreasing* addresses.

LDMDB, STMDB Decrement before

loading from *four less* than the named address and
storing into *decreasing* addresses.

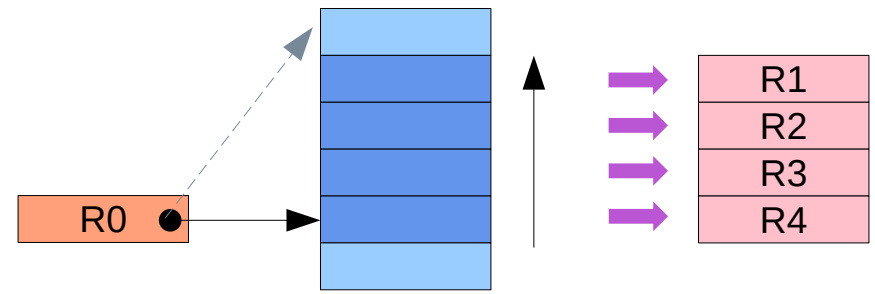
<http://www.cburch.com/books/arm/>

LDM / STM examples (9)

Across all four modes, the **highest-numbered** register always corresponds to the **highest address** in memory.

Thus, the instruction **LDMDA R0, { R1-R4 }** will place R4 into the address named by **R0**, **R3** into **R0 - 4**, and so on.

useful when we want to use a block of unused memory as a **stack**.



LDMDA, STMDA
Decrement after

<http://www.cburch.com/books/arm/>

PUSH, POP Synonyms

```
PUSH{cond} reglist  
POP{cond} reglist
```

Synonyms

```
PUSH = STMDB R13! = STMFD R13!  
POP = LDMIA R13! or even LDM = LDMFD R13!
```

Assume

the base register **SP (R13)**

the adjusted address **written back** to the base register

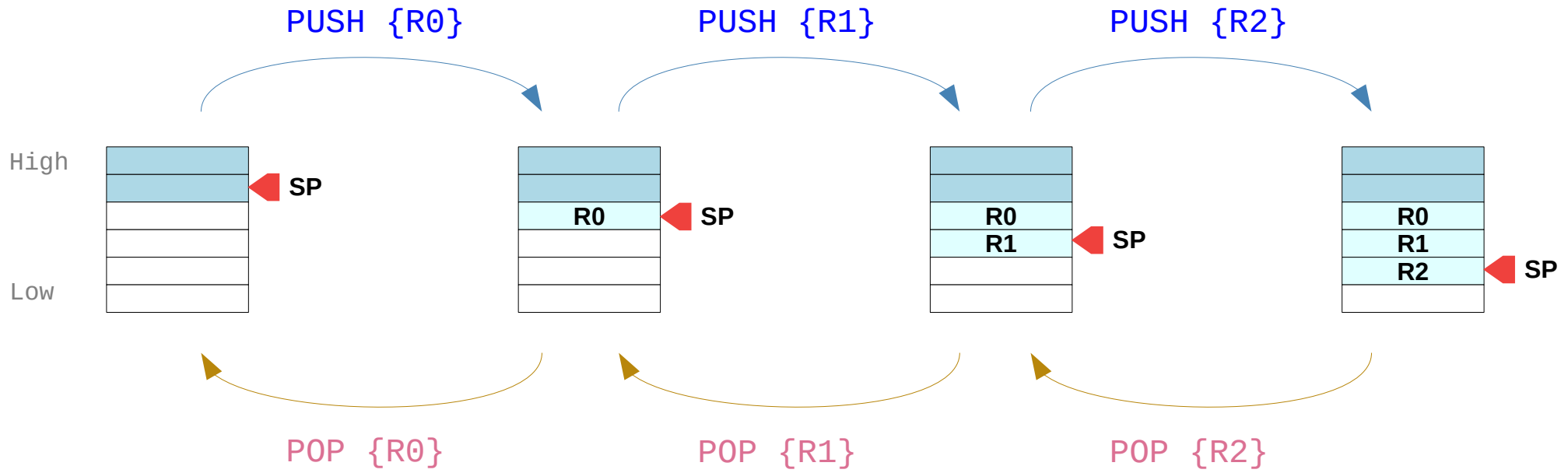
registers are stored on the stack in **numerical order**

with the lowest numbered register at the lowest address.

Full Descending Stack with SP (=R13)

PUSH, POP examples

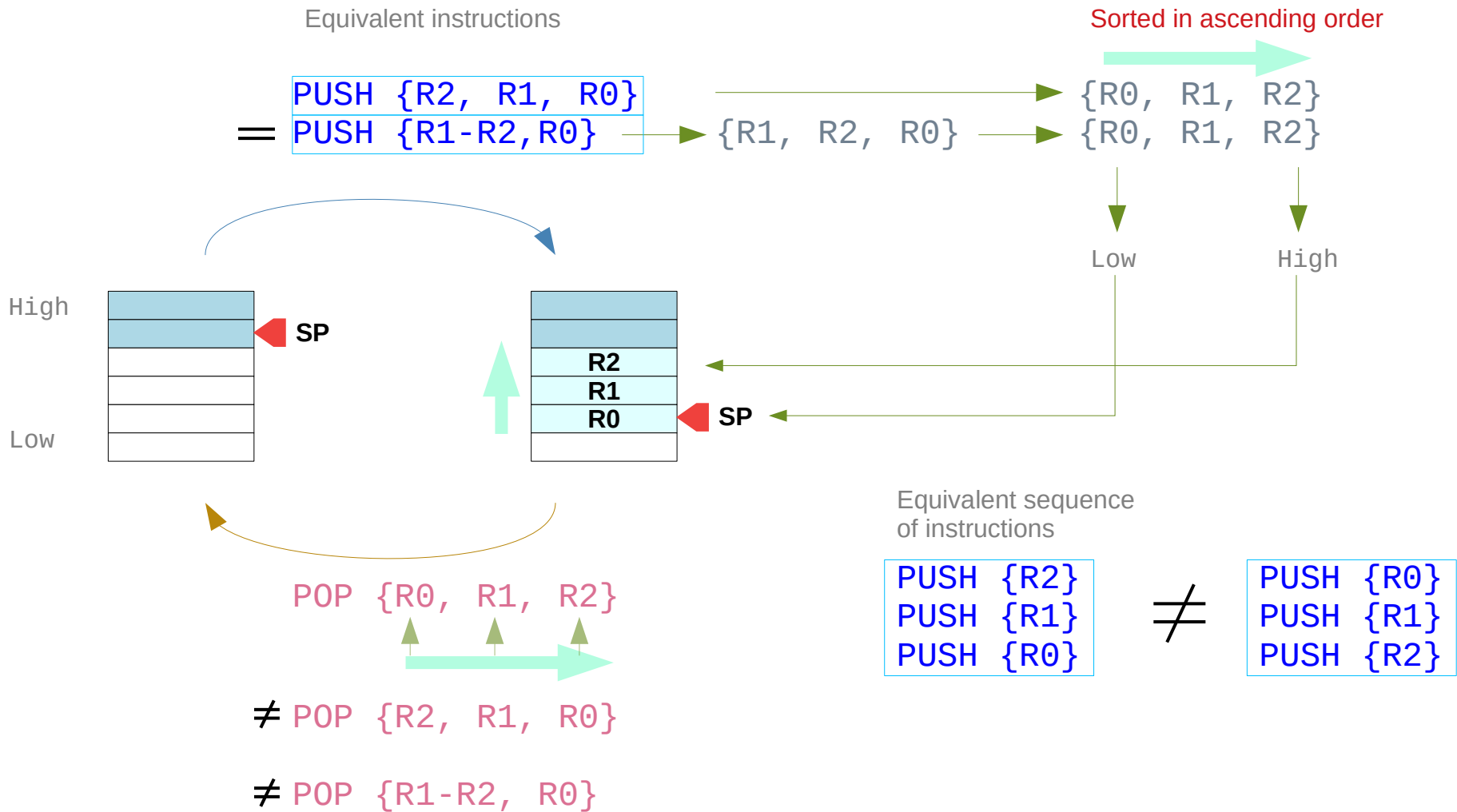
STMDB : Decrement *SP* Before **STR**



LDMIA : Increment *SP* After **LDR**

Full Descending Stack with *SP* (=R13)

Reglist examples



Stack Types and Stack Top Operations

Stack Types – Semantics

$(F,E) \times (A,D) = \{ FA, FD, EA, ED \}$ (Full, Empty) \times (Ascending, Descending)

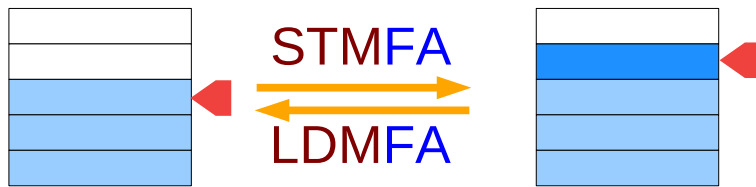
PUSH (STM) / POP (LDM)
over an $\{ FA / FD / EA / ED \}$ type stack

Stack Top Operations – Syntax

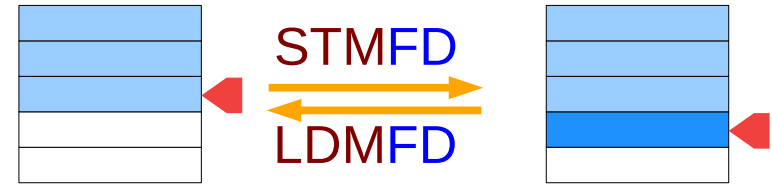
$(I,D) \times (B,A) = \{ IB, IA, DB, DA \}$ (Increment, Decrement) \times (After, Before)

$\{ Inc / Dec \}$ stack top operation
 $\{ Before / After \}$ STM / LDM

Stack Types



Full Ascending Stack

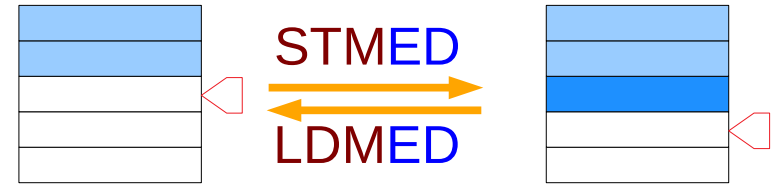


Full Descending Stack

Default Stack Type, SP (R13)



Empty Ascending Stack



Empty Descending Stack

(F_ / E_) and (_B / _A) reasoning

STMF□

If the stack top is **full**

STM□B

then inc / dec the stack pointer
before storing a new element

STME□

If the stack top is **empty**

STM□A

then inc / dec the stack pointer
after storing a new element

LDMF□

If the stack top is **full**

LDM□A

then inc / dec the stack pointer
after getting an element

LDME□

If the stack top is **empty**

LDM□B

then inc / dec the stack pointer
before getting an element

□ = { Ascend
Descend

□ = { Inc
Dec

(_A / _D) and (I_ / D_) reasoning

STM□**A** To push
onto the **ascending** stack

STMI□ **Increment** the stack top pointer

STM□**D** To push
onto the **descending** stack

STMD□ **Decrement** the stack top pointer

LDM□**A** To pop
from the **ascending** stack

LDMD□ **Decrement** the stack top pointer







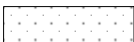

LDM□**D** To pop
from the **descending** stack

LDMI□ **Increment** the stack top pointer

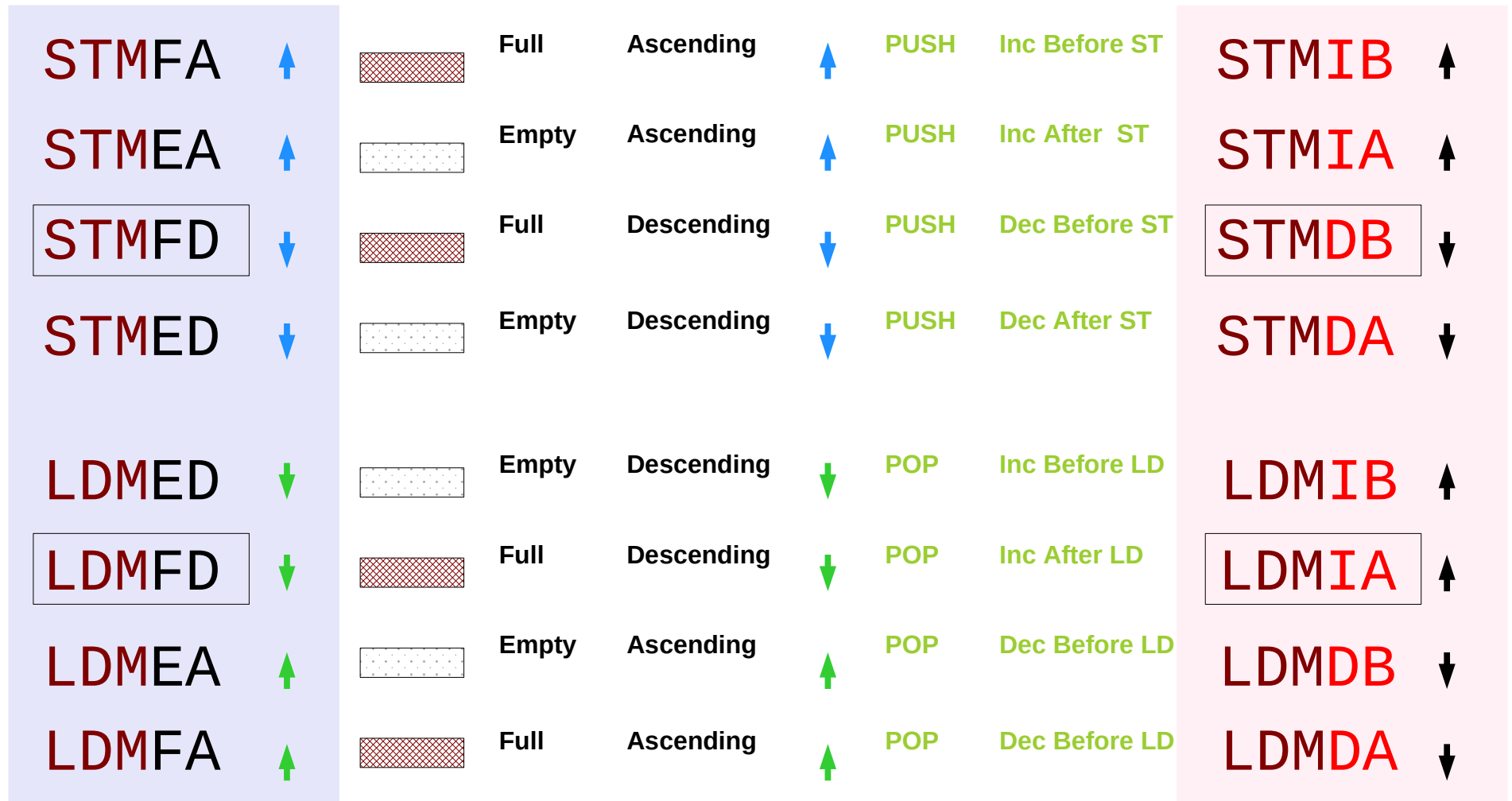
□ = { **Full**
Empty

□ = { **Before**
After

Block copy view → Stack view

STMIB	↑	PUSH	↑	Ascending	Inc Before ST		Full	STMFA	↑
STMIA	↑	PUSH	↑	Ascending	Inc After ST		Empty	STMFA	↑
STMDB	↓	PUSH	↓	Descending	Dec Before ST		Full	STMFD	↓
STMDA	↓	PUSH	↓	Descending	Dec After ST		Empty	STMED	↓
LDMIB	↑	POP	↓	Descending	Inc Before LD		Empty	LDMED	↓
LDMIA	↑	POP	↓	Descending	Inc After LD		Full	LDMFD	↓
LDMDB	↓	POP	↑	Ascending	Dec Before LD		Empty	LDMEA	↑
LMDA	↓	POP	↑	Ascending	Dec Before LD		Full	LDMFA	↑

Stack view → Block copy view



Stack View Addressing

STM**FA** r8! {r0, r1, r4}

LDM**FA** r8! {r0, r1, r4}

STM**EA** r8! {r0, r1, r4}

LD**ME**A r8! {r0, r1, r4}

STM**FD** r8! {r0, r1, r4}

LDM**FD** r8! {r0, r1, r4}

STM**ED** r8! {r0, r1, r4}

LD**ME**D r8! {r0, r1, r4}

PUSH(STM) / POP(LDM)
on an **FA** type stack

PUSH(STM) / POP(LDM)
on an **EA** type stack

PUSH(STM) / POP(LDM)
on an **FD** type stack

PUSH(STM) / POP(LDM)
on an **ED** type stack

Stack Types – Semantics

Block Copy Addressing

STM**IB** r8! {r0, r1, r4}

Do **inc** stack top operation
before STM / LDM

LDM**IB** r8! {r0, r1, r4}

STM**IA** r8! {r0, r1, r4}

Do **inc** stack top operation
after STM / LDM

LDM**IA** r8! {r0, r1, r4}

STM**DB** r8! {r0, r1, r4}

Do **dec** stack top operation
before STM / LDM

LDM**DB** r8! {r0, r1, r4}

STM**DA** r8! {r0, r1, r4}

Do **dec** stack top operation
after STM / LDM

LDM**DA** r8! {r0, r1, r4}

Stack Top Operations – Syntax

Addressing mode examples (4)

References

- [1] <ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf>
- [2] <https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf>