Data Transfer (4A)

Young Won Lim
2/6/20

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

This document was produced by using LibreOffice.

Young Won Lim
2/6/20

Based on

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

Memory Objects

Memory object type	Register	Example operand
Constants in code space	PC	`=Constant  [PC, #28]`
Local variables on the stack	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)
```

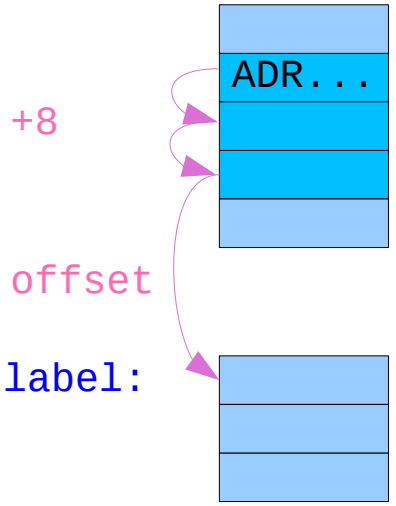
https://stackoverflow.com/questions/42065155/how-to-replace-ldr-with-adr-in-assembler

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
```

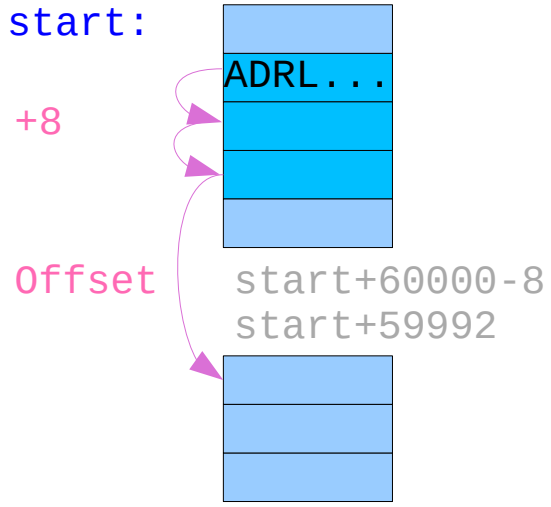
https://stackoverflow.com/questions/42065155/how-to-replace-ldr-with-adr-in-assembler

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
```

+4

```
; 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

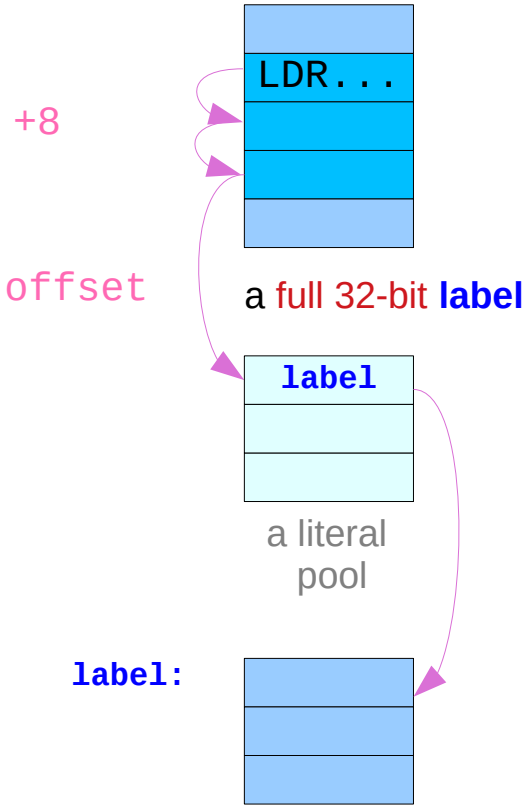
```
LDR {cond} Rd, =label        (=address)
```

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-assembler

LDR Rd, =number

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

placing the **number** in a literal pool

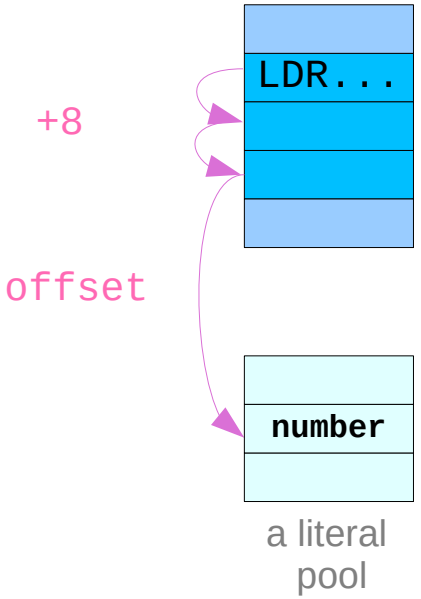
```
LDR Rd,=number
```

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

the offset to a literal pool

```
MOV Rd, #imm16
```

for a small range number,



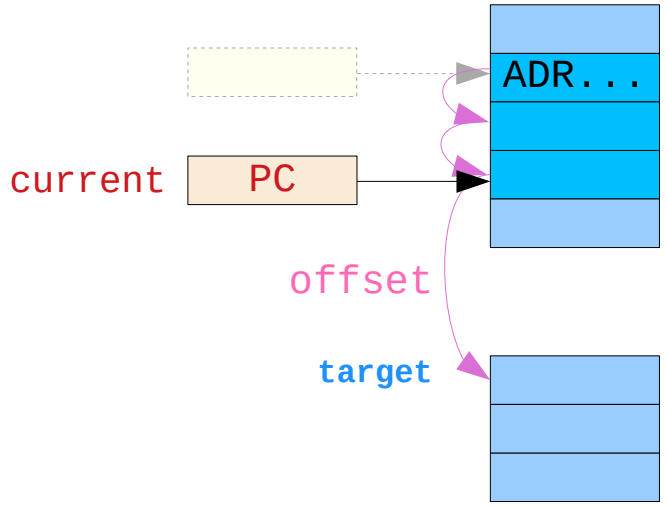
https://stackoverflow.com/questions/42065155/how-to-replace-ldr-with-adr-in-assembler

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)

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



pc – target = offset

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
    adrl    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

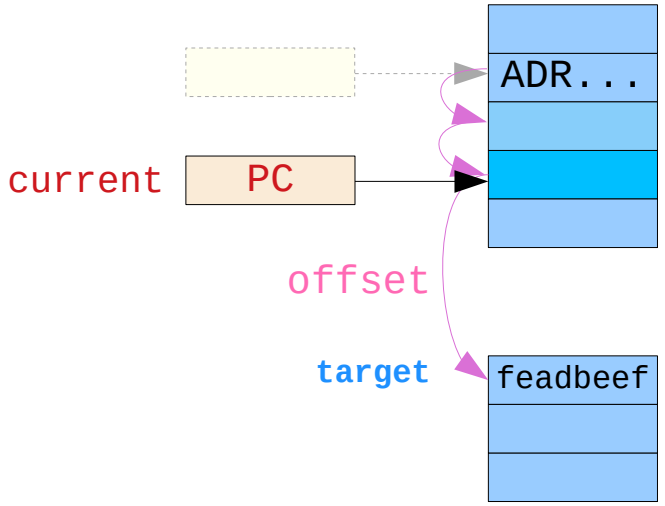
https://stackoverflow.com/questions/15774581/getting-an-label-address-to-a-register-on-arm

ADR / ADRL offsets

```
target:
        .long 0xfeadbeef

    adr     r0, target
    adrl    r0, target
```

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



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

```
r0 ← pc – target
r0 ← offset
```

pc – target = offset

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

ADR vs ADRL

The difference between **`adr`** and **`adrl`**
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 (**`adrl`**)

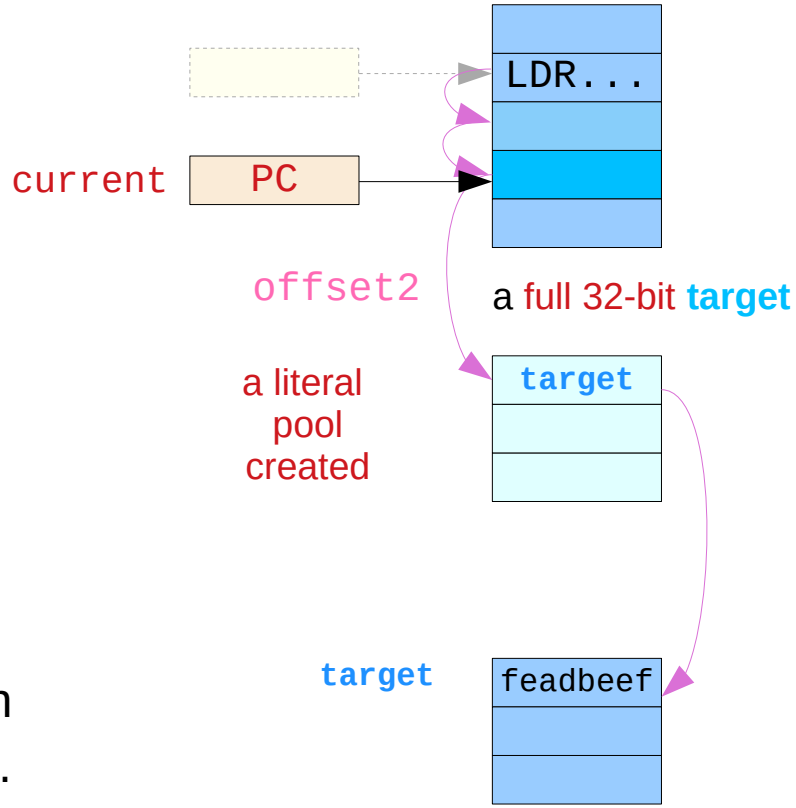
`adrl` will usually be faster than
the **`ldr`** variant (**`ldr =target`**)
which get a full 32-bits address in the literal pool

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).

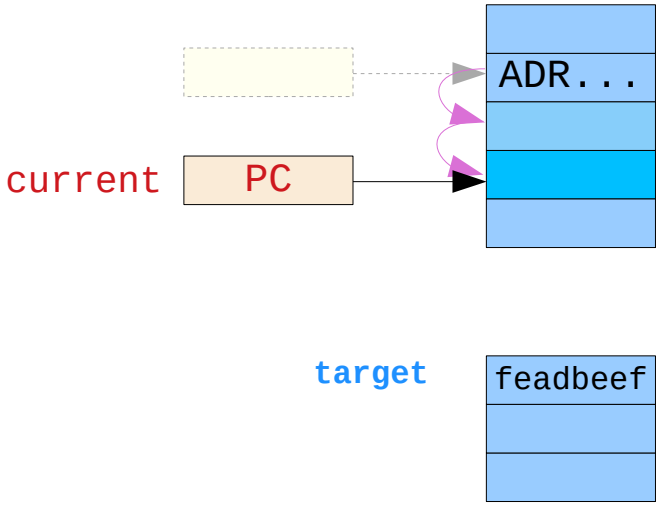
https://stackoverflow.com/questions/15774581/getting-an-label-address-to-a-register-on-arm

Manual computing an address

```
target:
     .long 0xfeadbeef

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

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



`.` : 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

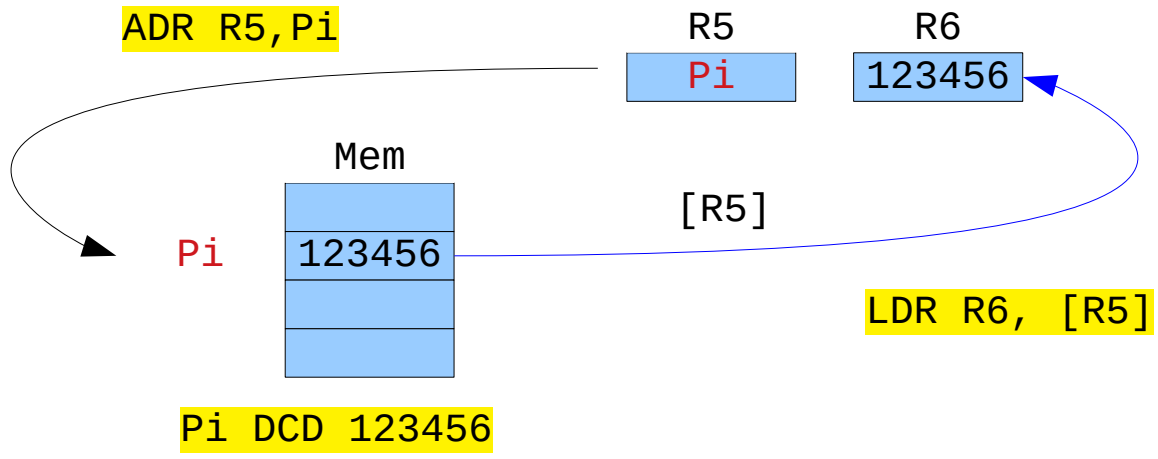
https://stackoverflow.com/questions/15774581/getting-an-label-address-to-a-register-on-arm

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

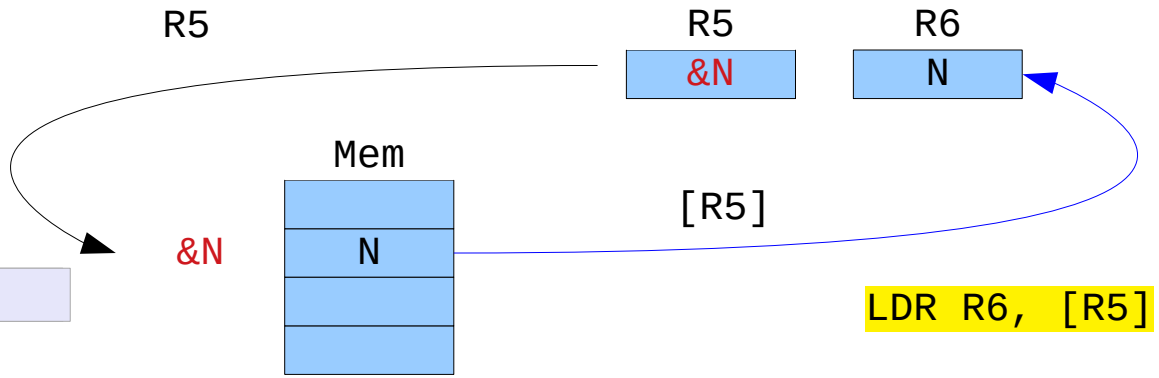
R5 ← PC-relative Pi



LDR =number example

```
In    LDR R5, =0x12345678
      LDR R6, [R5]
      …
      BX LR
```

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




```
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

PUSH, POP Synonyms

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

Synonyms

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

Assume

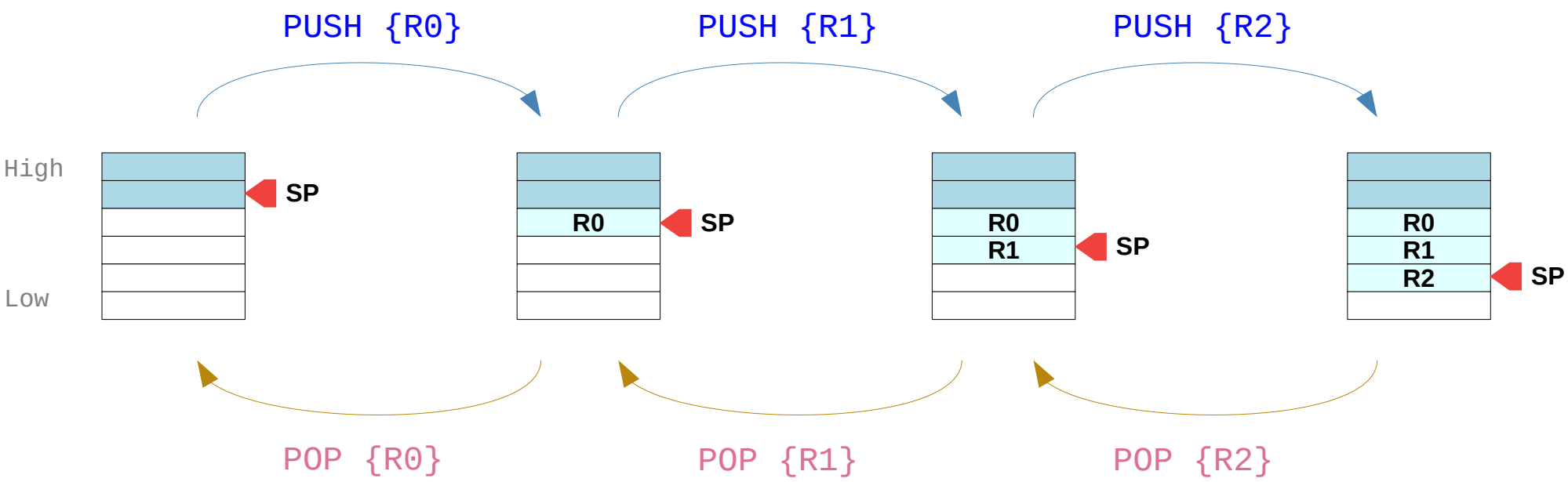
the base register SP (R13)
the adjusted address written back to the base register

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

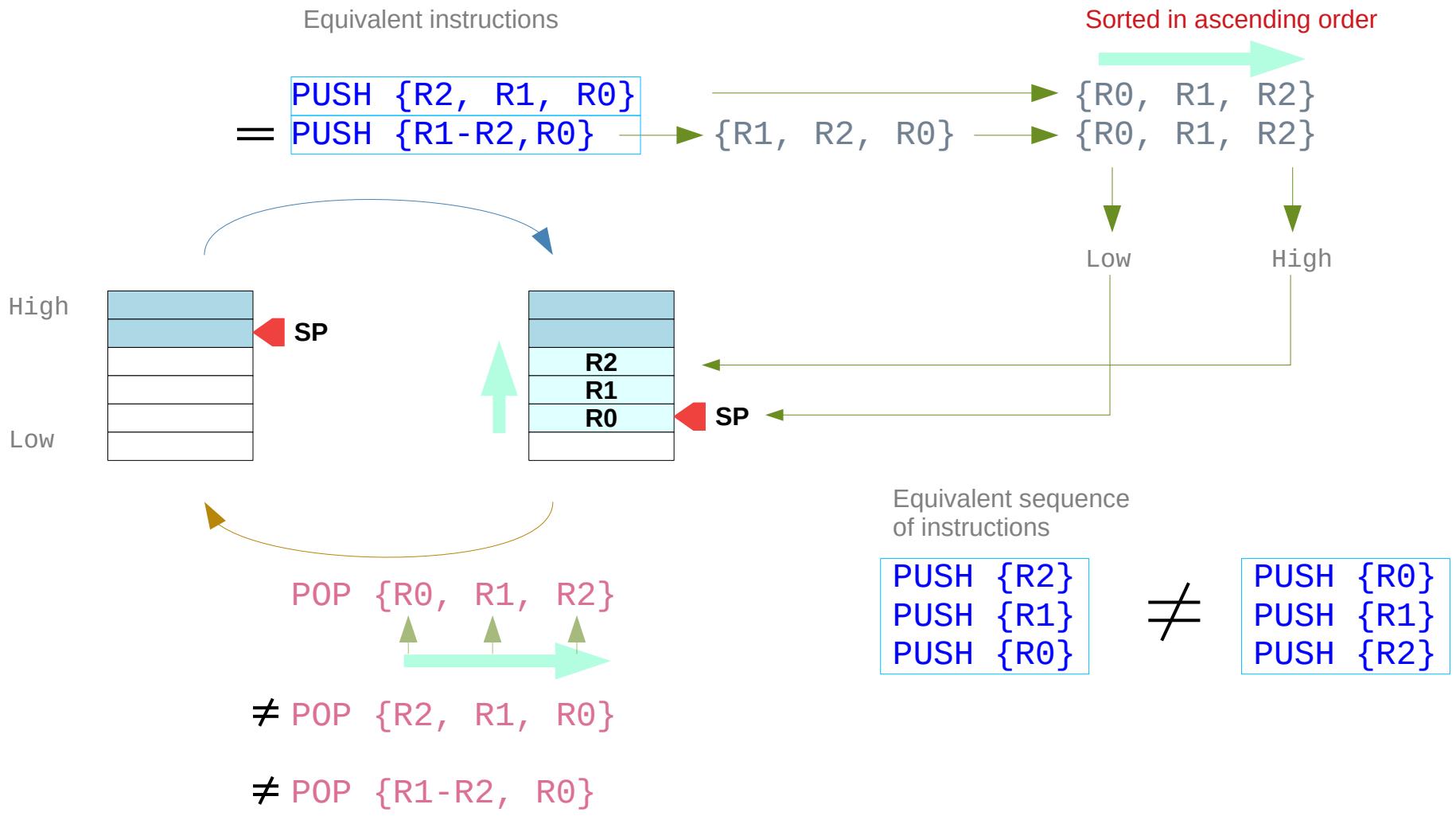
STMDB : Decrement *SP* Before STR



LDMIA : Increment *SP* After LDR

Full Descending Stack with SP (=R13)

Reglist examples



Stack Types and Stack Top Operations

Stack Types – Semantics

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

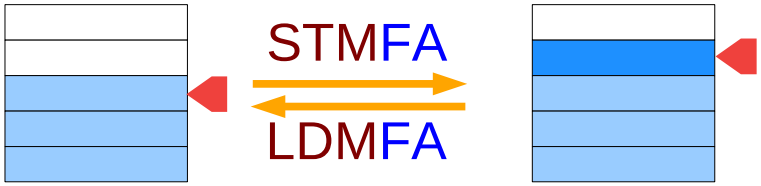
PUSH (STM) / POP (LDM)
over an { FA / FD / EA / ED } type stack

Stack Top Operations – Syntax

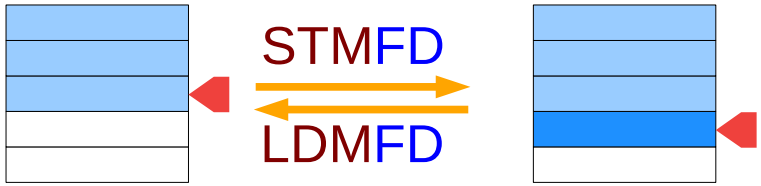
(I,D) x (B,A) = { IB, IA, DB, DA } (Increment, Decrement) x (After, Before)

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

Stack Types



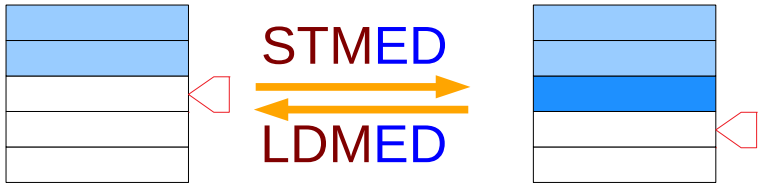
Full Ascending Stack



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

□ = { **A**scend, **D**escend

□ = { **I**nc, **D**ec

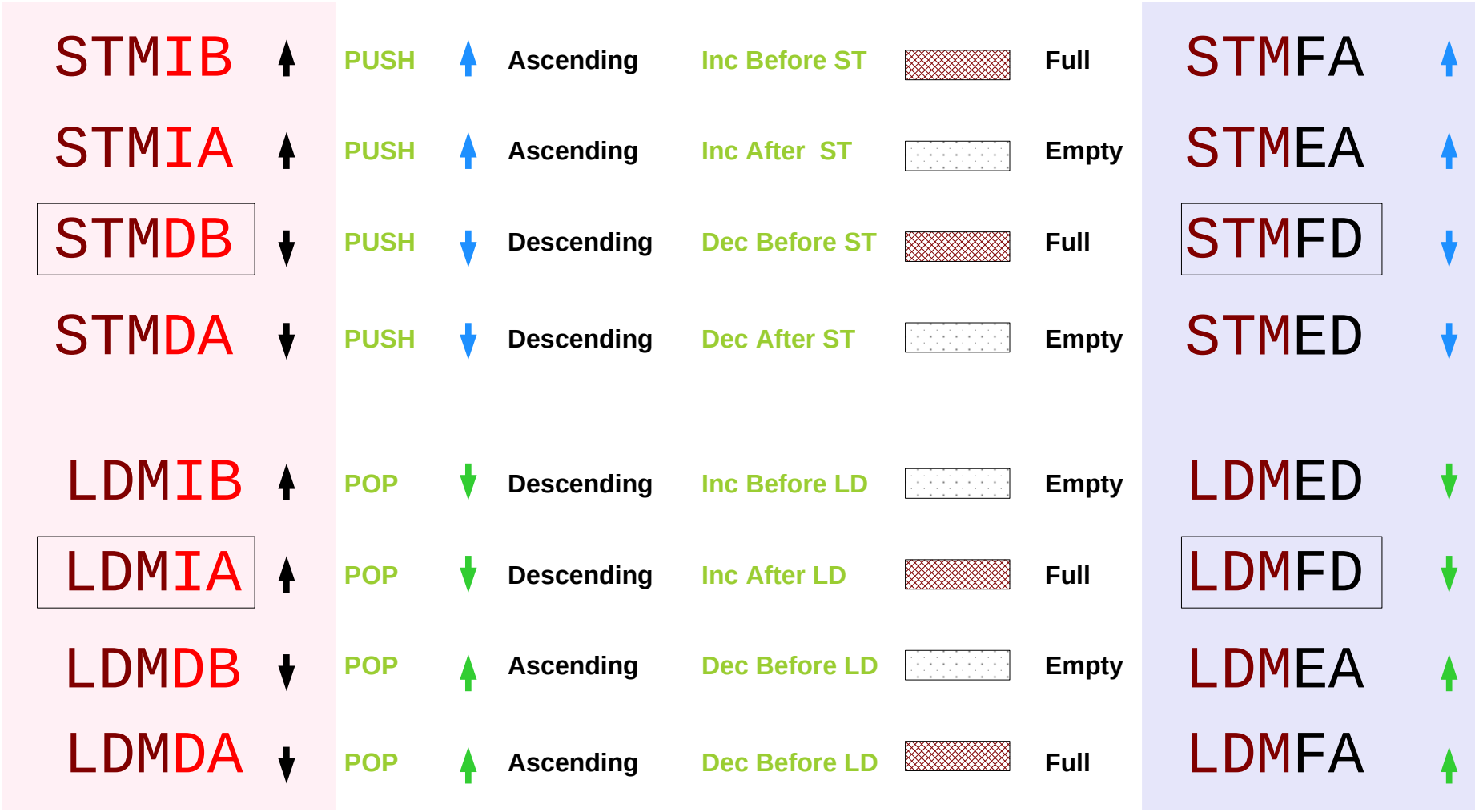
(_A / _D) and (I_ / D_) reasoning

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

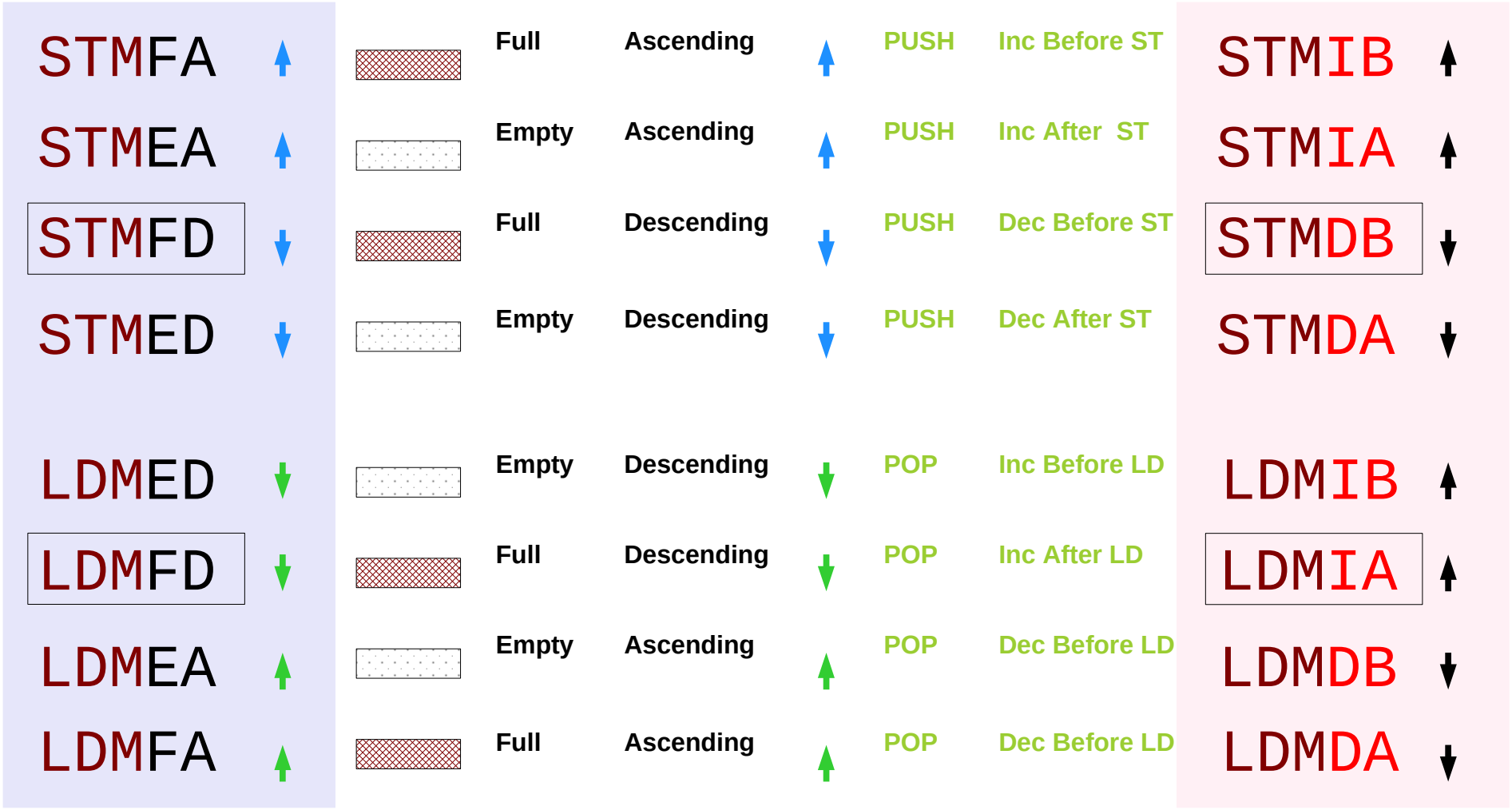
□ = { **F**ull, **E**mpty

□ = { **B**efore, **A**fter

Block copy view ➡ Stack view



Stack view ➡ Block copy view



Stack View Addressing

```
STMFA r8! {r0,r1,r4}
LDMFA r8! {r0,r1,r4}
```

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

```
STMEA r8! {r0,r1,r4}
LDMEA r8! {r0,r1,r4}
```

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

```
STMFD r8! {r0,r1,r4}
LDMFD r8! {r0,r1,r4}
```

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

```
STMED r8! {r0,r1,r4}
LDMED r8! {r0,r1,r4}
```

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

Stack Types – Semantics

Block Copy Addressing

`STMIB r8! {r0,r1,r4}` `LDMIB r8! {r0,r1,r4}`	Do **inc** stack top operation **before** STM / LDM
`STMIA r8! {r0,r1,r4}` `LDMIA r8! {r0,r1,r4}`	Do **inc** stack top operation **after** STM / LDM
`STMDB r8! {r0,r1,r4}` `LDMDB r8! {r0,r1,r4}`	Do **dec** stack top operation **before** STM / LDM
`STMDA r8! {r0,r1,r4}` `LDMDA r8! {r0,r1,r4}`	Do **dec** stack top operation **after** STM / LDM

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