ISA Addressing Mode (2A)

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

This document was produced by using LibreOffice.

Based on

ARM System-on-Chip Architecture, 2nd ed, Steve Furber

Registers

MRS {<cond>}	**Rd**, CPSR
MRS {<cond>}	**Rd**, SPSR
<op> {<cond>} {S}	**Rd**, **Rn**, #<32-bit immediate>
LDR|STR {<cond>} {B}	**Rd**, [**Rn**, <offset>] {!}
LDR|STR {<cond>} {B} {T}	**Rd**, [**Rn**], <offset>
LDR|STR {<cond>} {B}	**Rd**, LABEL
LDR|STR {<cond>} H|SH|SB	**Rd**, [**Rn**, <offset>] {!}
LDR|STR {<cond>} H|SH|SB	**Rd**, [**Rn**], <offset>
LDM|STM {<cond>} <add mode>	**Rn**{!}, <registers>
<op> {<cond>} {S}	**Rd**, **Rn**, **Rm**, {<shift>}
MSR {<cond>}	CPSR_<field>, **Rm**
MSR {<cond>}	SPSR_<field>, **Rm**
MUL {<cond>} {S}	**Rd**, **Rm**, **Rs**
MLA {<cond>} {S}	**Rd**, **Rm**, **Rs**, **Rn**
<mul> {<cond>} {S}	**RdHi**, **RdLo**, **Rm**, **Rs**
SWP {<cond>} {B}	**Rd**, **Rm**, [**Rn**]
CLZ {<cond>}	**Rd**, **Rm**

Registers in Data Processing Instructions

Rn	1st Operand Reg
Rm	2nd Operand Reg
Rd	Source / Destination Reg

Rd	←	**Rn**		1st Operand + Immediate
Rd	←	**Rn**	**Rm**	1st Operand + 2nd Operand

Registers in Data Transfer Instructions

Rn 1st Operand Reg / Base Reg

Rm 2nd Operand Reg / Operand Reg /Offset Reg / Source Reg

Rd Source / Destination Reg

Rd ← [**Rn**] **Load** dst reg from base reg

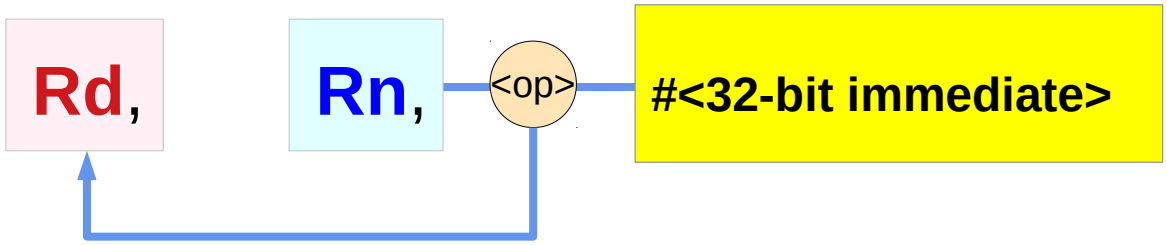
Rd → [**Rn**] **Store** src reg to base reg

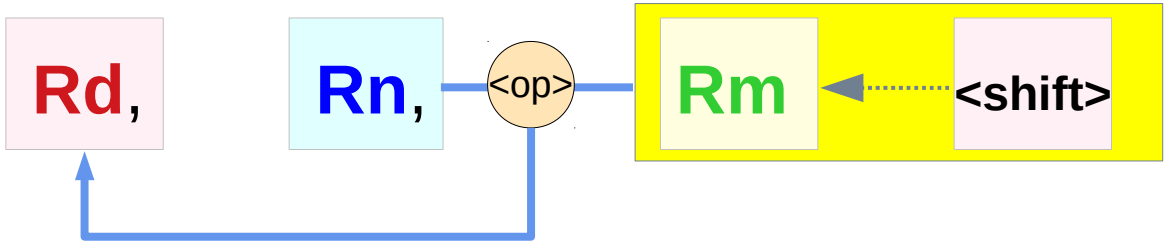
[**Rn**] ← **Reg list** **Store Multiple** list of regs to base reg

[**Rn**] → **Reg list** **Load Multiple** list of regs from base reg

Registers in data processing

<op> {<cond>} {S} **Rd**, **Rn**, **#<32-bit immediate>**
<op> {<cond>} {S} **Rd**, **Rn**, **Rm**, {**<shift>**}



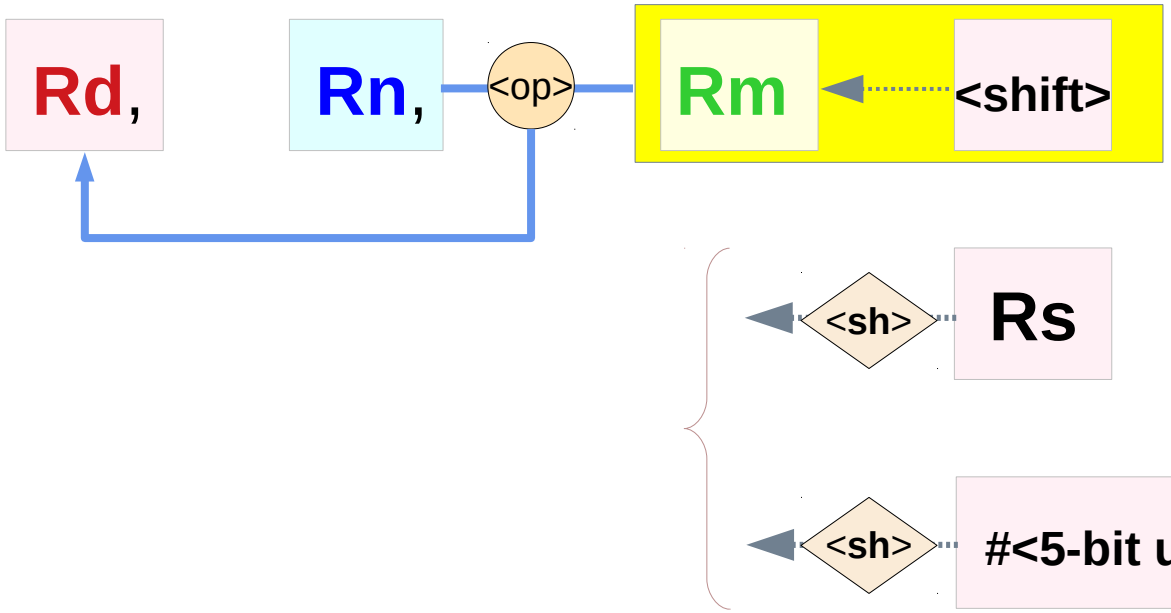


a shifted operand

```
ADD  r3, r2, r1,   LSL #3     ; r3 := r2 + r1 * 2^3
ADD  r5, r5, r3,   LSL r2     ; r5 := r5 + r3 * 2^r2
MOV r0, r0,        LSR #2     ; r0 :=      r0 / 2^2
```

Registers in data processing with a shifted operand

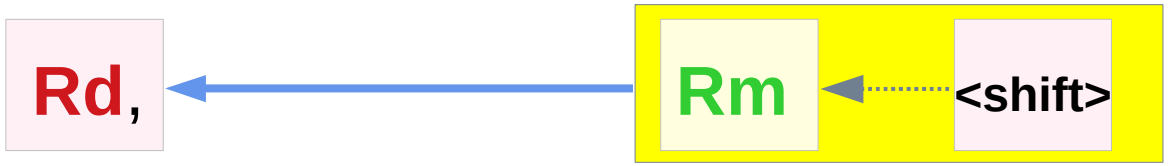
<op> {<cond>} {S} **Rd**, **Rn**, **Rm**, {**<shift>**}



sh	Shift Type
00	Logical Left
01	Logical Right
10	Arithmetic Right
11	Rotate Right

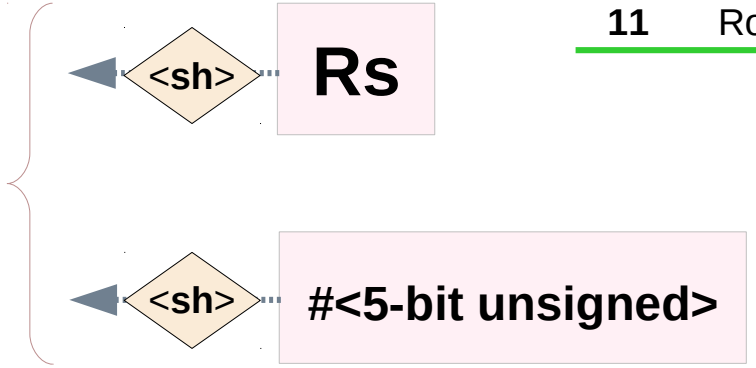
Registers in stand alone shift instructions

<op> {<cond>} {S}	**Rd**, **Rm**, **Rs**
<op> {<cond>} {S}	**Rd**, **Rm**, **#n**



Sh	Shift Type
00	Logical Left
01	Logical Right
10	Arithmetic Right
11	Rotate Right

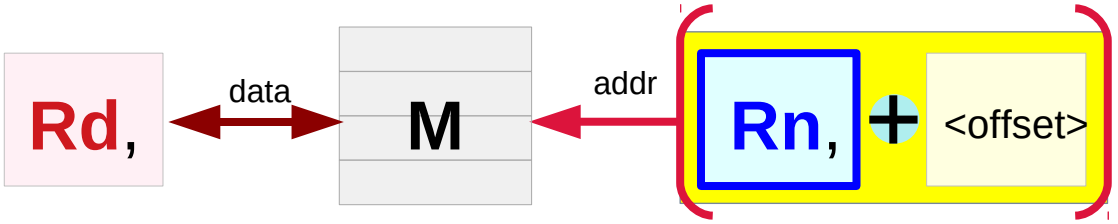
<op>	Shift Type
LSL	Logical Left
LSR	Logical Right
ASR	Arithmetic Right
ROR	Rotate Right



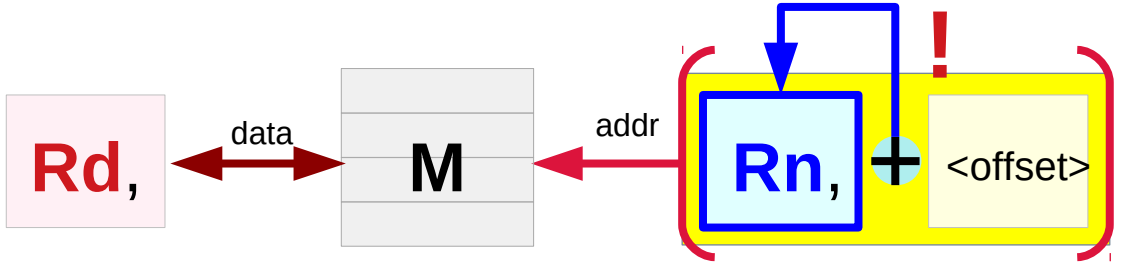
RRX {<cond>} {S}	**Rd**, **Rm**

Registers in data transfer

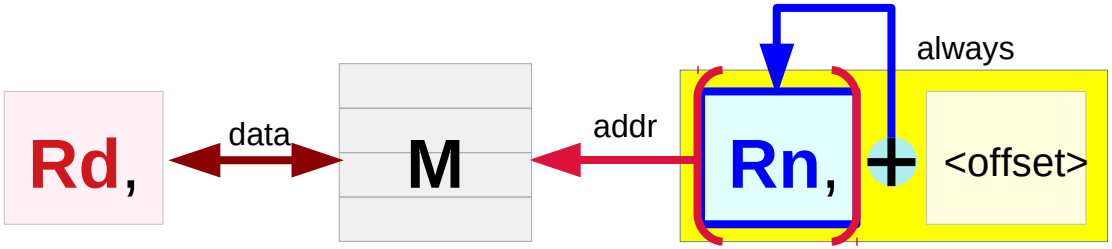
LDR|STR {<cond>} {B}	Rd, [Rn, <offset>] {!}
LDR|STR {<cond>} {B} {T}	Rd, [Rn], <offset>
LDR|STR {<cond>} {B}	Rd, LABEL



pre-indexed



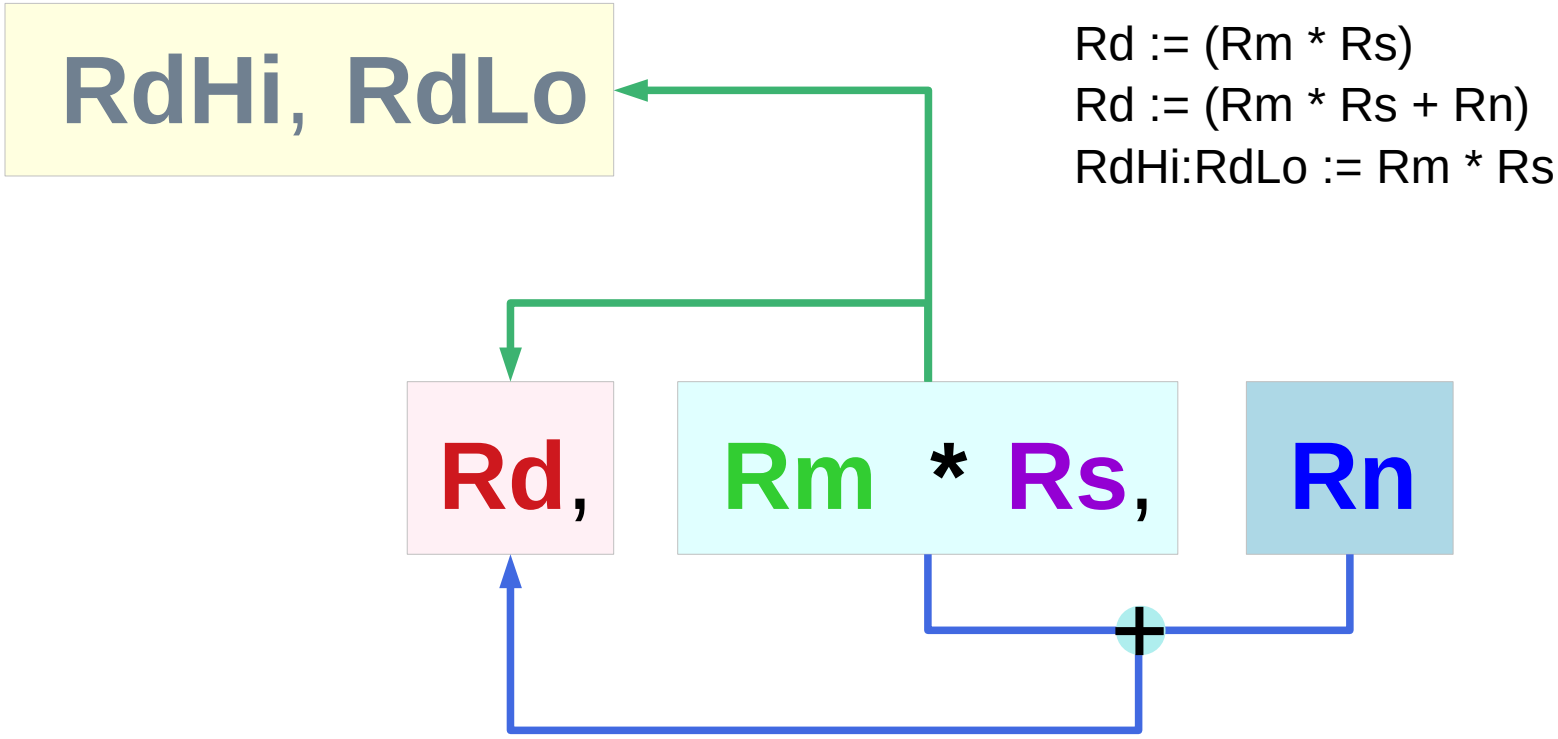
pre-indexed.
with autoindex !



post-indexed

Registers in multiplication

MUL {<cond>} {S}	**Rd**, **Rm**, **Rs**
MLA {<cond>} {S}	**Rd**, **Rm**, **Rs**, **Rn**
<mul> {<cond>} {S}	**RdHi**, **RdLo**, **Rm**, **Rs**



Status registers

MRS {<cond>}	**Rd**, CPSR
MRS {<cond>}	**Rd**, SPSR
MSR {<cond>}	CPSR_<field>, **Rm**
MSR {<cond>}	SPSR_<field>, **Rm**

M R ⬅ S

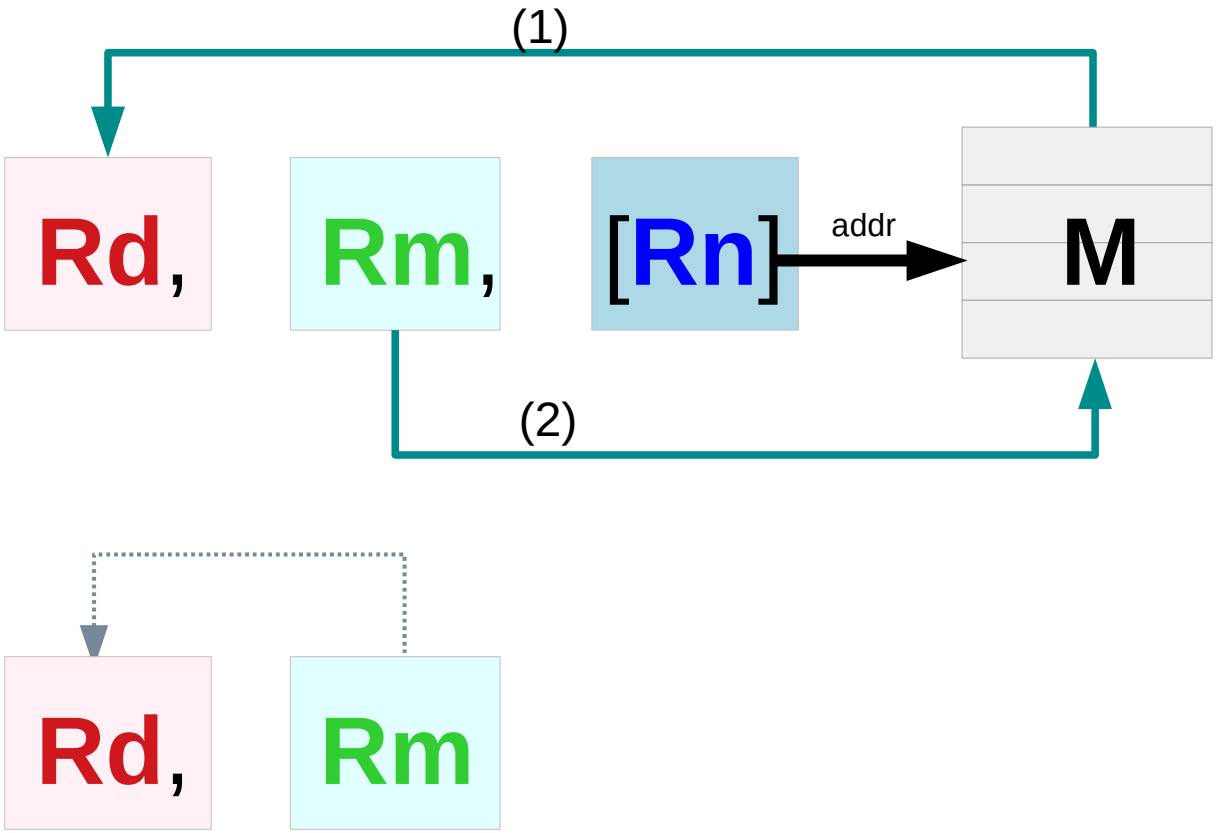
Rd, CPSR
Rd, SPSR

M S ⬅ R

CPSR_f **Rm**
SPSR_f **Rm**

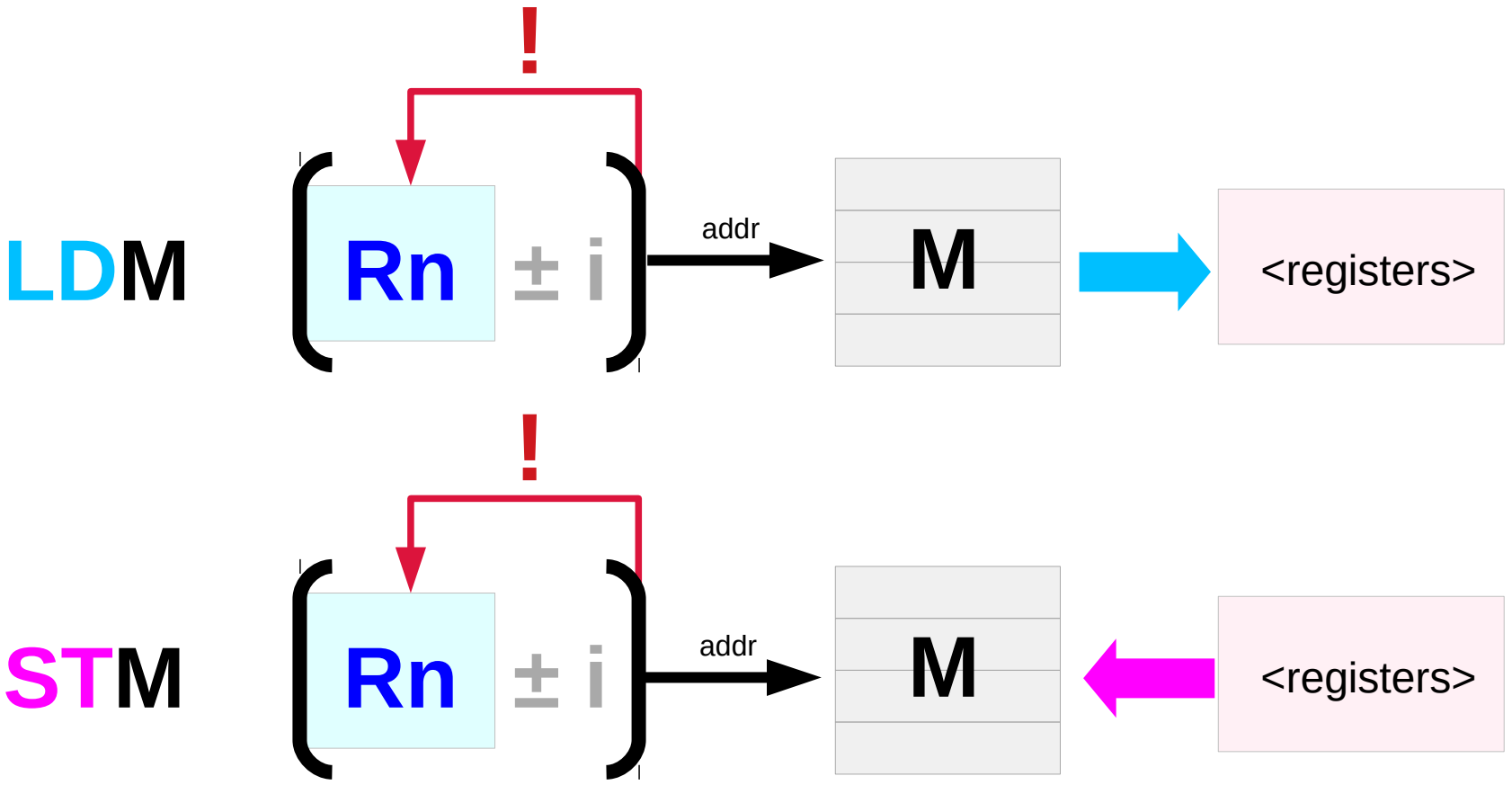
Swap and clear zero registers

SWP {<cond>} {B}	**Rd**, **Rm**, [**Rn**]
CLZ {<cond>}	**Rd**, **Rm**



Registers in Multiple Data Transfers

LDM|STM {<cond>} <add mode> **Rn**{!}, <registers>



General Addressing

4-address instruction

```
    ADD     d, s1, s2, next_i       ; d := s1 + s2
```

3-address instruction

```
    ADD     d, s1, s2               ; d := s1 + s2
```

2-address instruction

```
    ADD     d, s1                   ; d := d + s1
```

1-address instruction

```
    ADD     s1                      ; acc := acc + s1
```

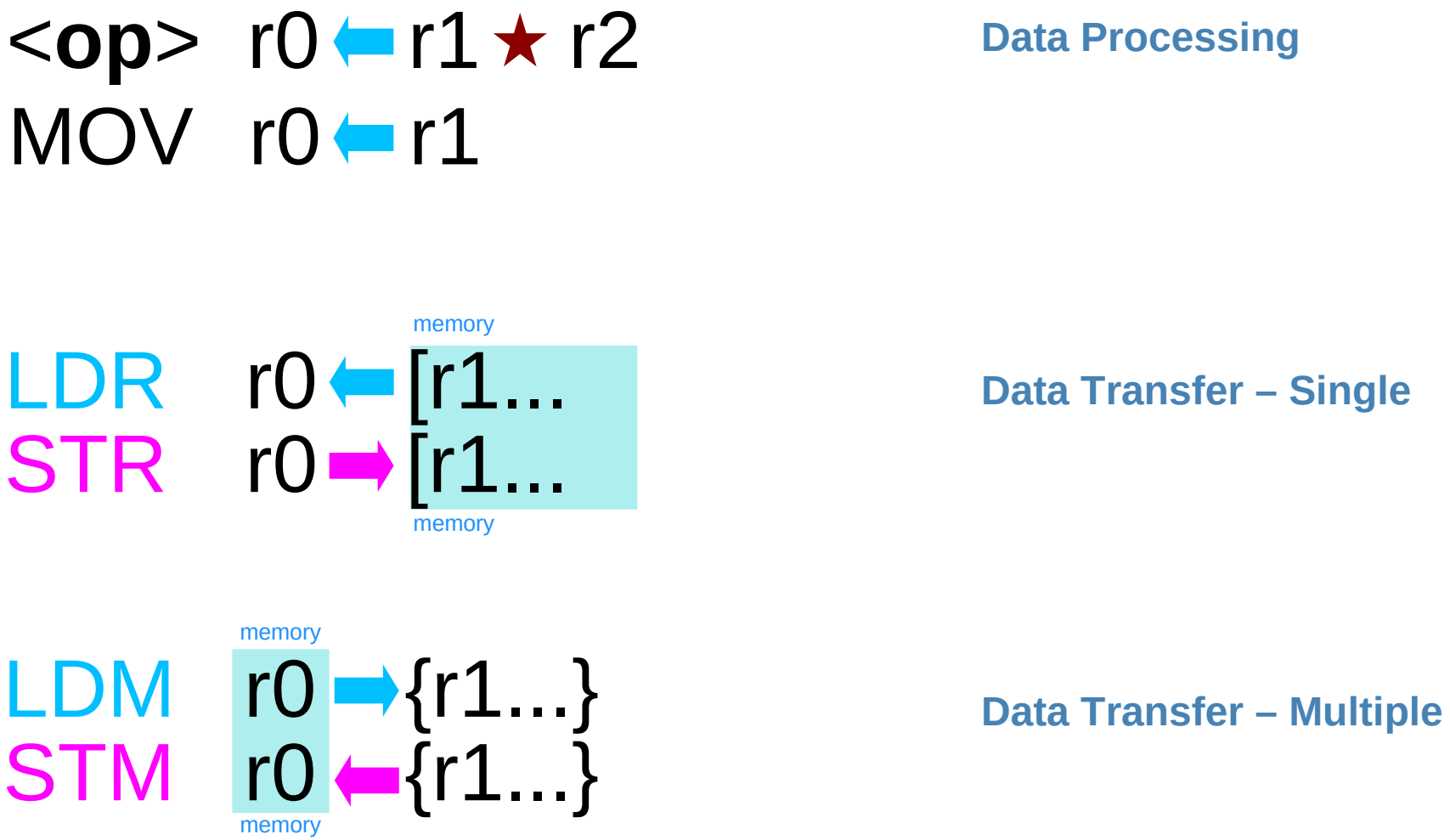
0-address instruction

```
    ADD                             ;  top := top + next  of the stack
```

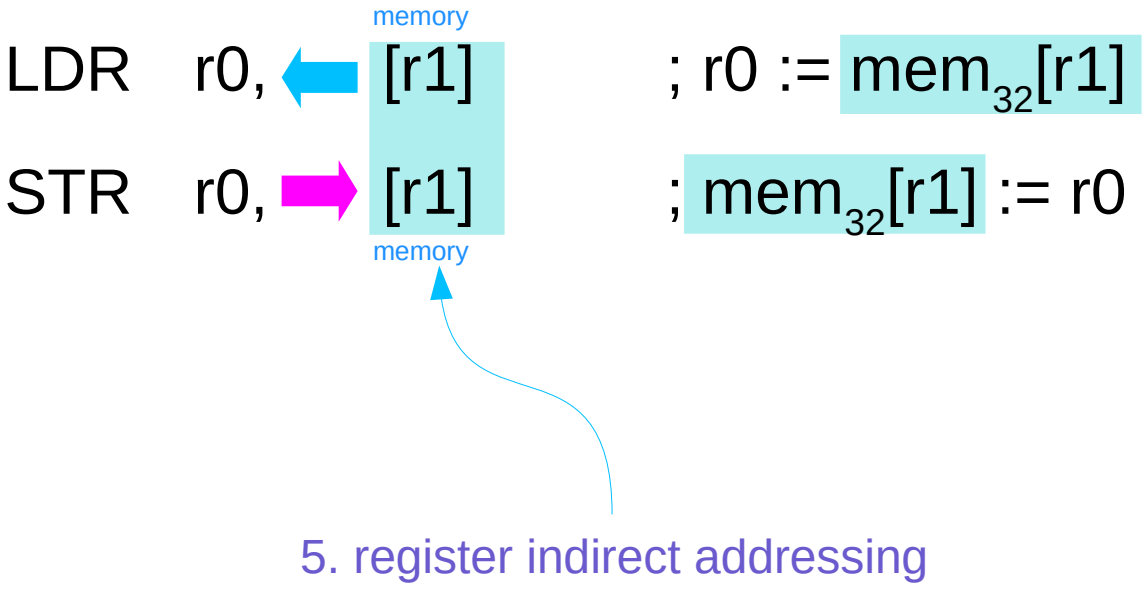
General Addressing Modes

1. immediate addressing	a binary value
2. absolute addressing	a full binary address
3. indirect addressing	address of a desired value
4. register addressing	a desired value in a register
5. register indirect addressing	address in a register
6. base plus offset addressing	binary offset
7. base plus index addressing	offset in a register (index)
8. base plus scaled index addressing	multiplied by a constant
9. stack addressing	

General Addressing Modes



Single register load and store instructions



Base plus offset addressing

pre-indexed

LDR r0, [r1, #4] ; r0 := mem_{32}[r1 + 4]

pre-indexed, auto-index

LDR r0, [r1, #4] ! ; r0 := mem_{32}[r1 + 4]

; r1 := r1 + 4

post-indexed

LDR r0, [r1], #4 ; r0 := mem_{32}[r1]

; r1 := r1 + 4

6. base plus offset addressing

base : r1, offset #4

Pre-indexed Addressing

pre-indexed

LDR r0, [r1, #4] ; r0 := mem_{32}[r1 + 4]

[base, offset] compute first, then transfer

pre-indexed, auto-index

LDR r0, [r1, #4] **!** ; r0 := mem_{32}[r1 + 4]

; r1 := r1 + 4

[base, offset] compute first, then transfer

and then update base register

Post-indexed Addressing

post-indexed

```
LDR     r0,   [r1], #4        ; r0 := mem32[r1]
                              ; r1 := r1 + 4
```

using [base], transfer, then update base always

no **!**

My notation

LDR r0, [r1, #4] ; r0 := mem_{32}[r1 + 4] **pre-indexed**

[r1 + 4]

LDR r0, [r1, #4] ! ; r0 := mem_{32}[r1 + 4] **pre-indexed, auto-index**

[r1 += 4] ; r1 := r1 + 4

LDR r0, [r1], #4 ; r0 := mem_{32}[r1] **post-indexed**

[r1] += 4 ; r1 := r1 + 4

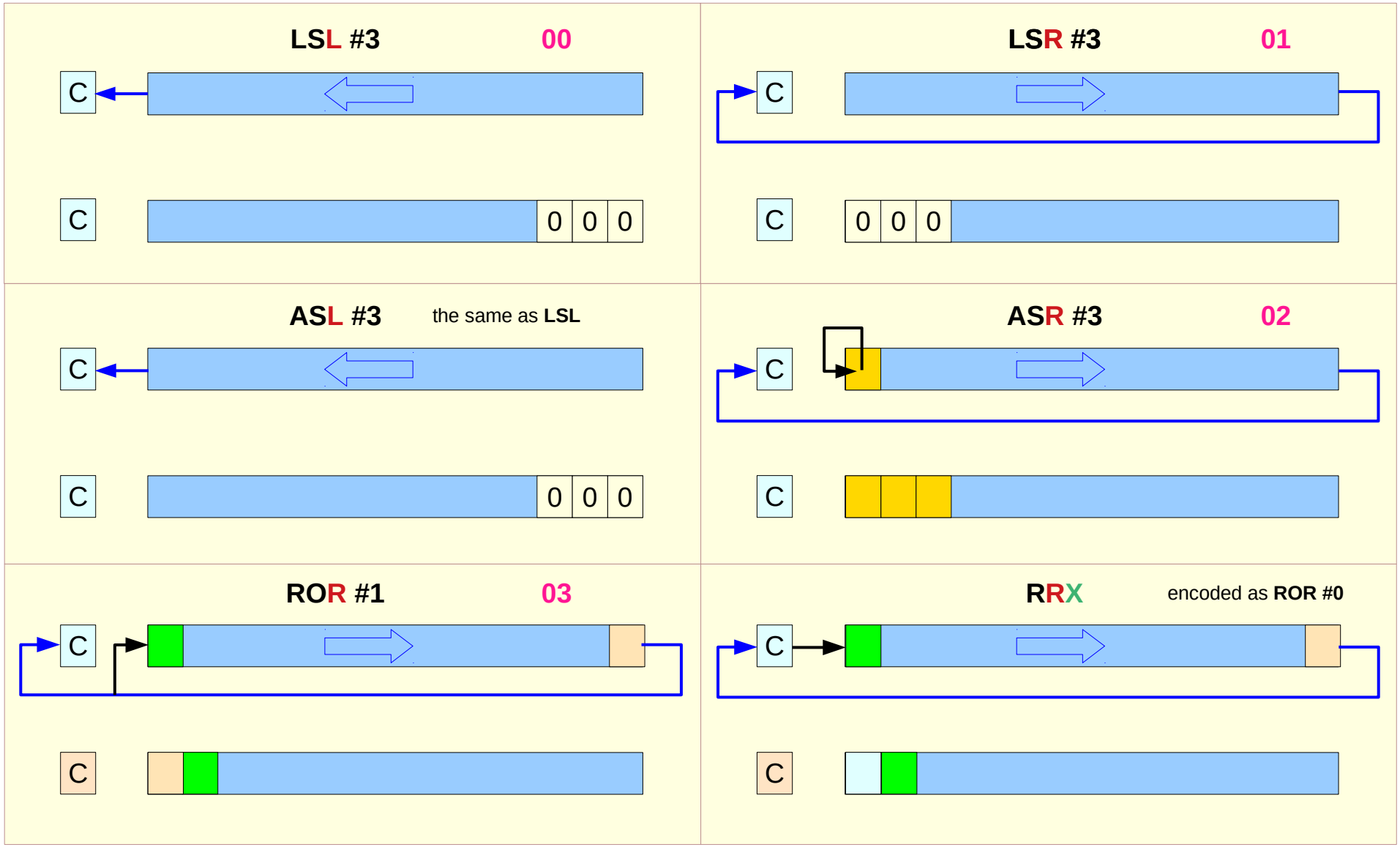
Shifted register operands

```
ADD     r3, r2, r1, LSL #3      ; r3 := r2 + r1*2^3
ADD     r5, r5, r3, LSL r2      ; r5 := r5 + r3*2^r2
```

LSL	00	Logical Shift Left
LSR	01	Logical Shift Right
ASL		Arithmetic Shift Left
ASR	10	Arithmetic Shift Right
ROR	11	Rotate Right
RRX		Rotate Right eXtended

Sh	Shift Type
00	Logical Left
01	Logical Right
10	Arithmetic Right
11	Rotate Right

Shift Types


LSL #3
00
C
C
0 0 0
LSR #3
01
C
C
0 0 0
ASL #3
the same as LSL
C
C
0 0 0
ASR #3
02
C
C
ROR #1
03
C
C
RRX
encoded as ROR #0
C
C

Stack 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

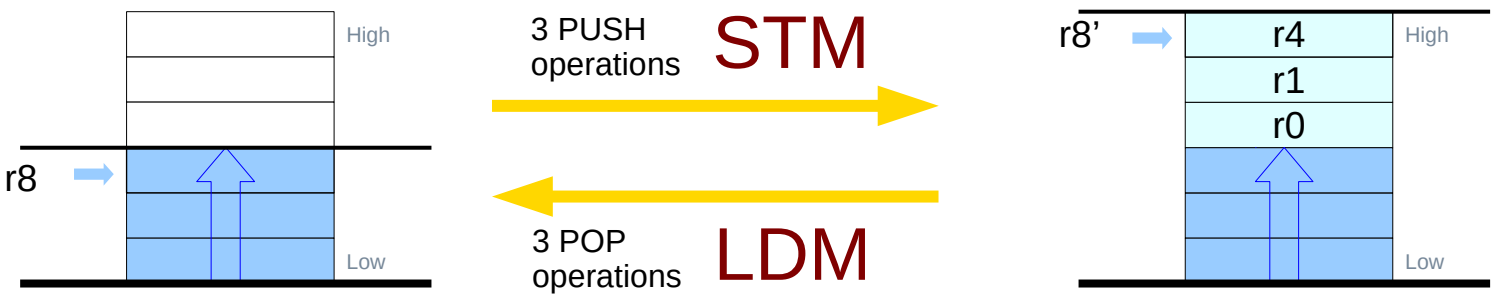
Stack Top Operations – Syntax

Ascending / Descending Stack

Full Ascending Stack	STMFA
	LDMFA
Empty Ascending Stack	STMEA
	LDMEA
Full Descending Stack	STMFD
	LDMFD
Empty Descending Stack	STMED
	LDMED

Ascending Stack – Store / Load

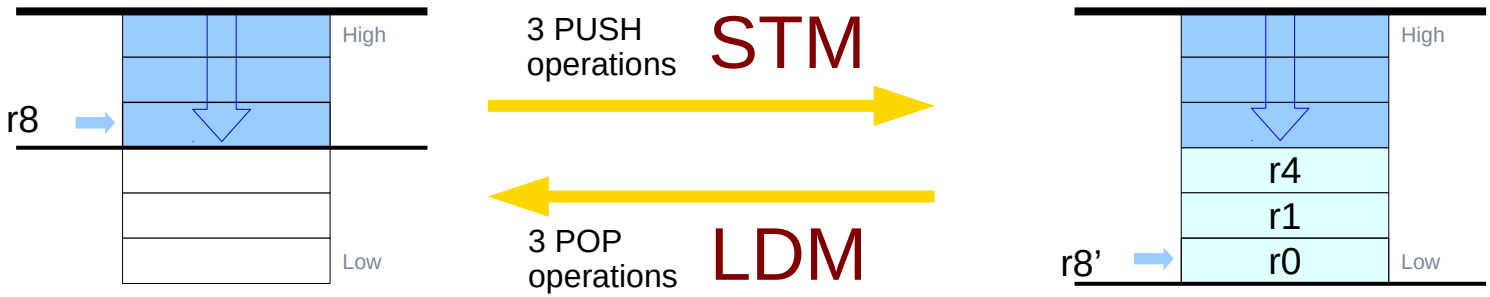
Full
Ascending Stack



Empty
Ascending Stack

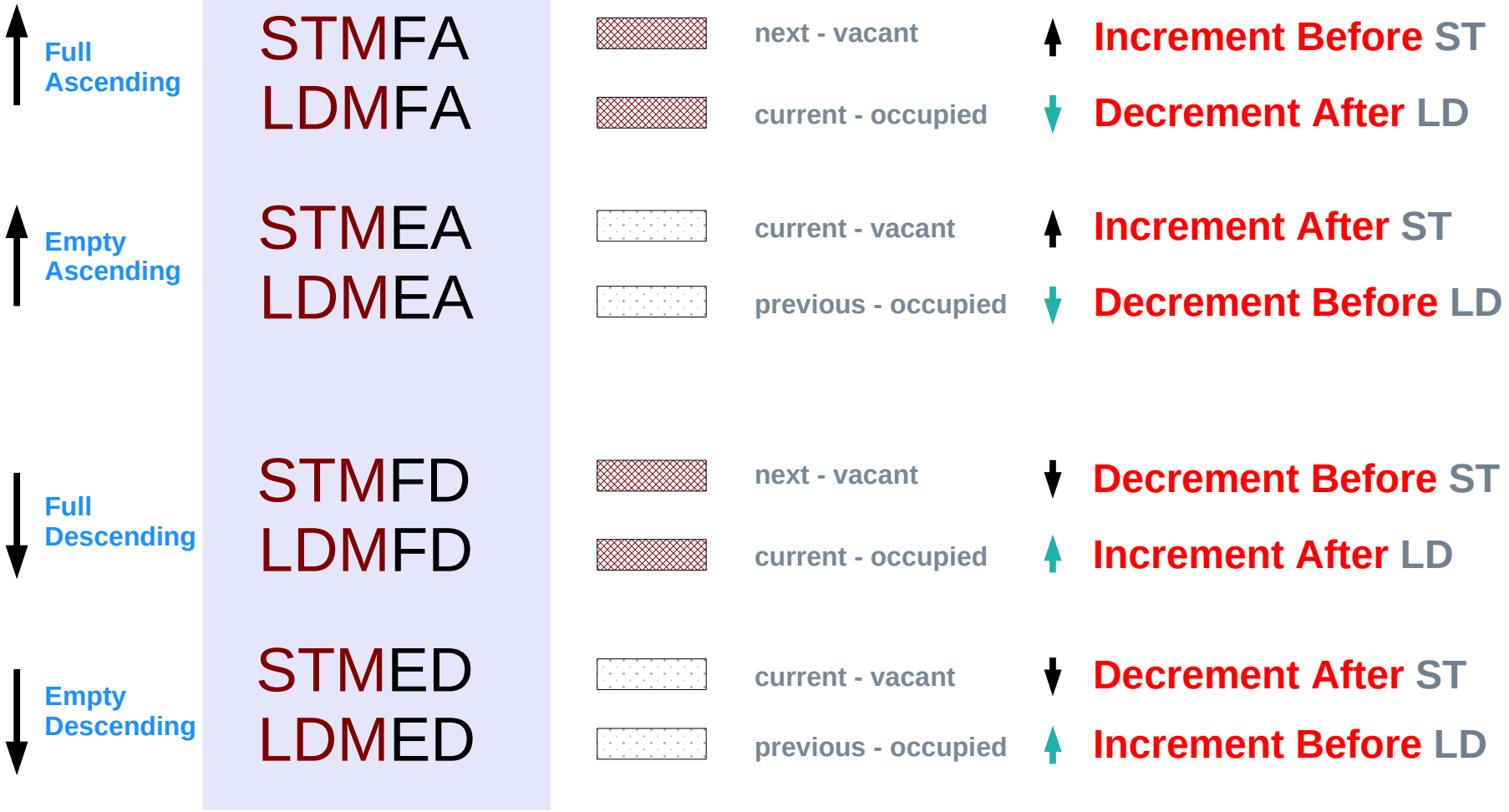
Descending Stack – Store / Load

Full Descending Stack



Empty Descending Stack

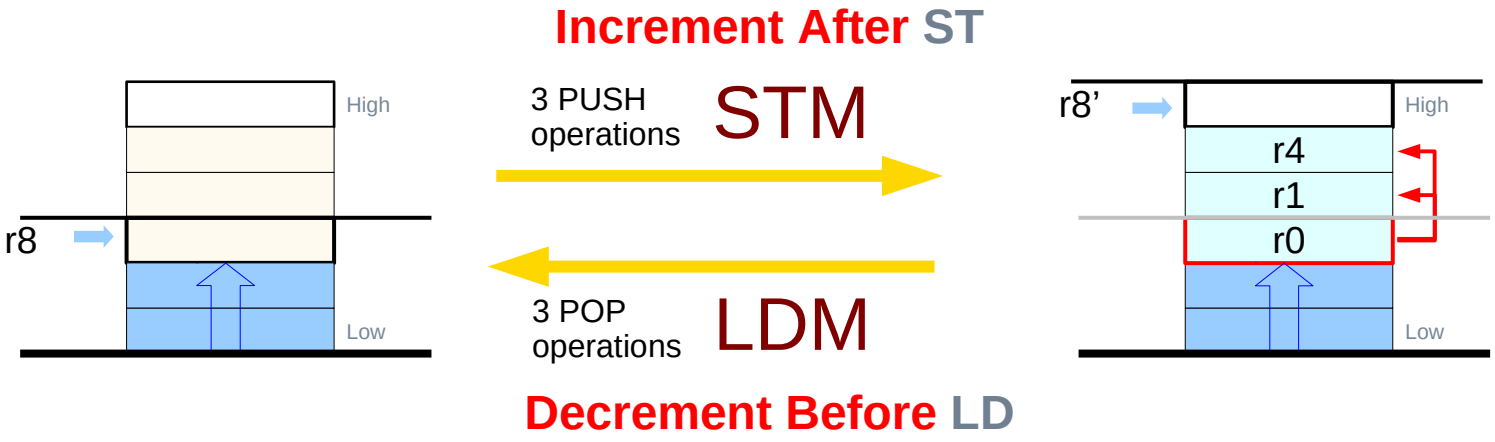
Ascending / Descending Stack & Increment / Decrement



Ascending stack – Increment / Decrement

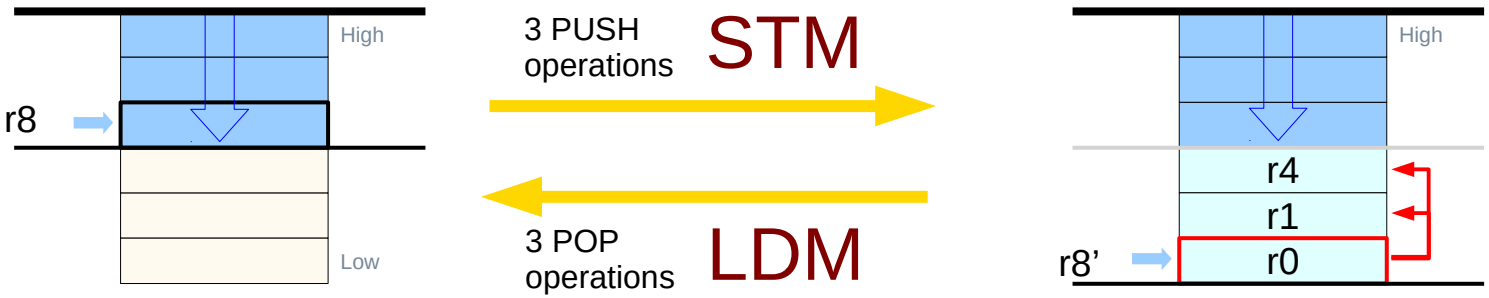
Full
Ascending Stack

Empty
Ascending Stack

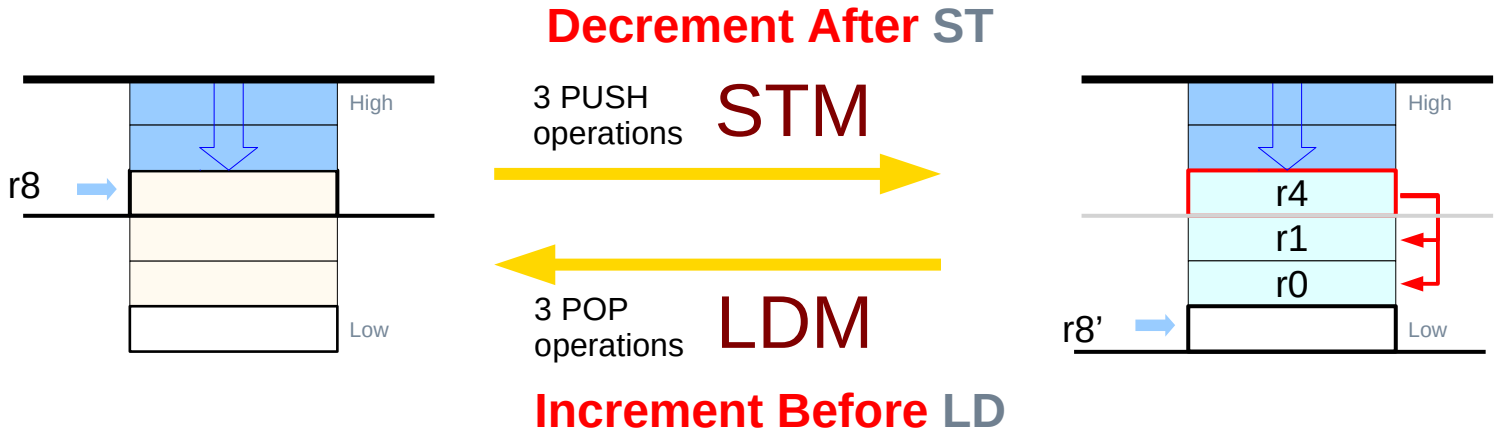


Descending Stack – Increment / Decrement

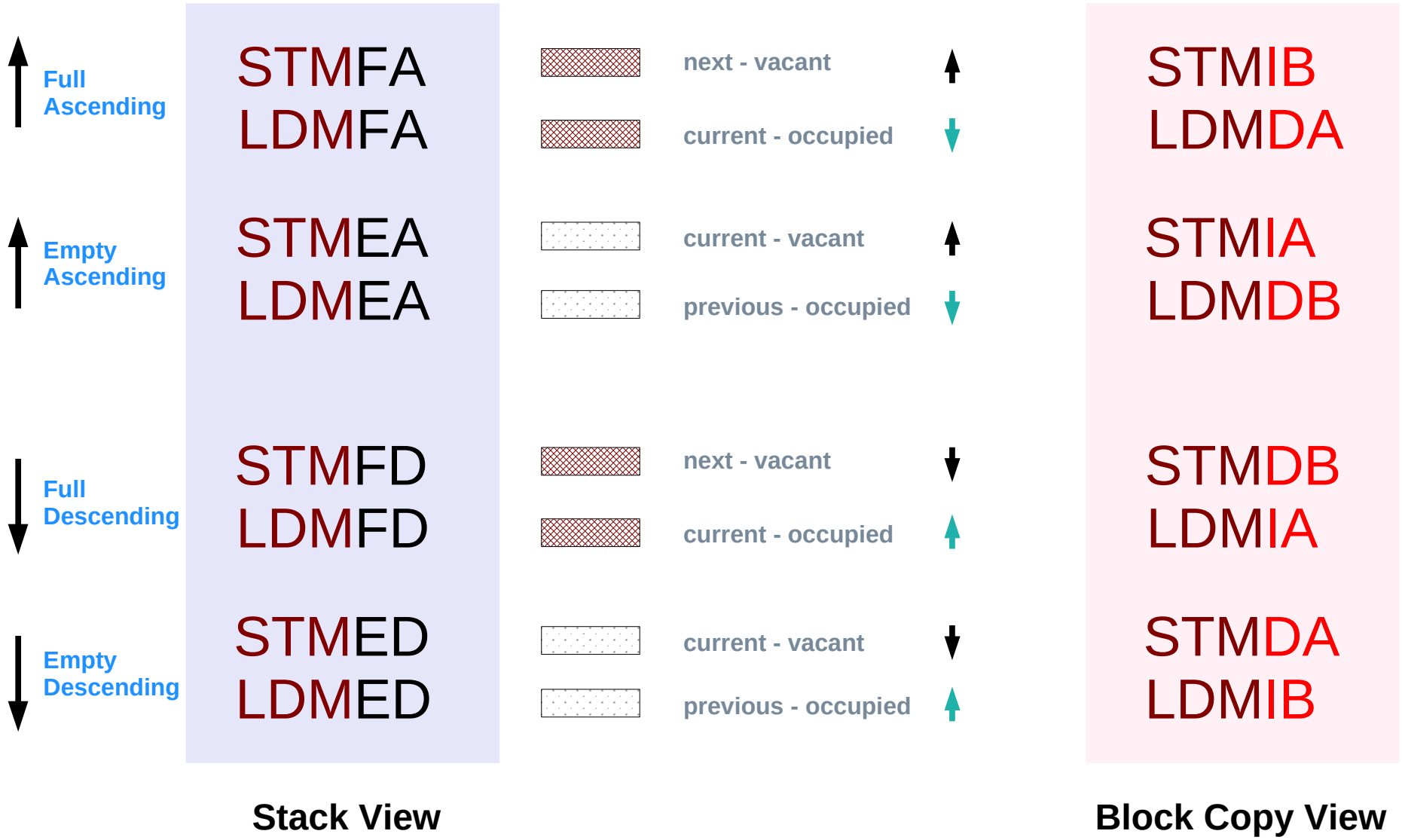
Full
Descending Stack



Empty
Descending Stack

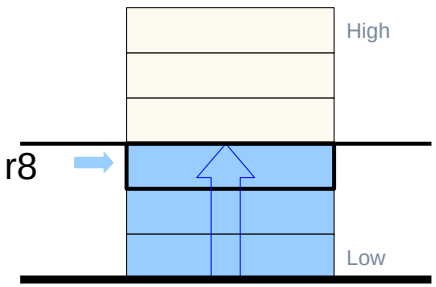


Stack and Block Copy Views



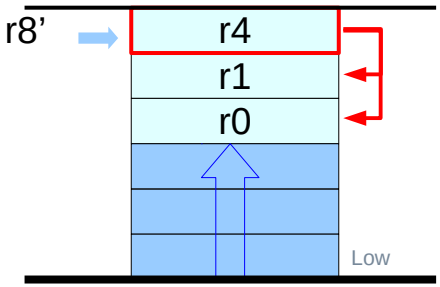
Ascending Stack – Equivalent Operations

Full
Ascending Stack

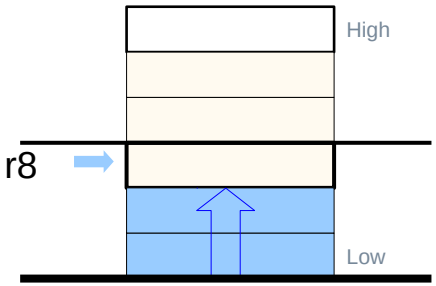


STMIB
STMFA

LDMDA
LDMFA

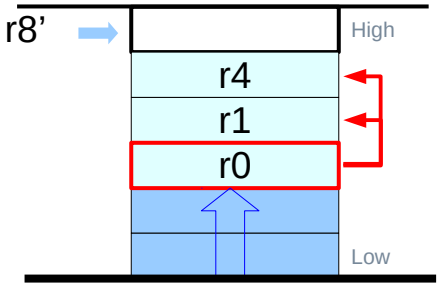


Empty
Ascending Stack



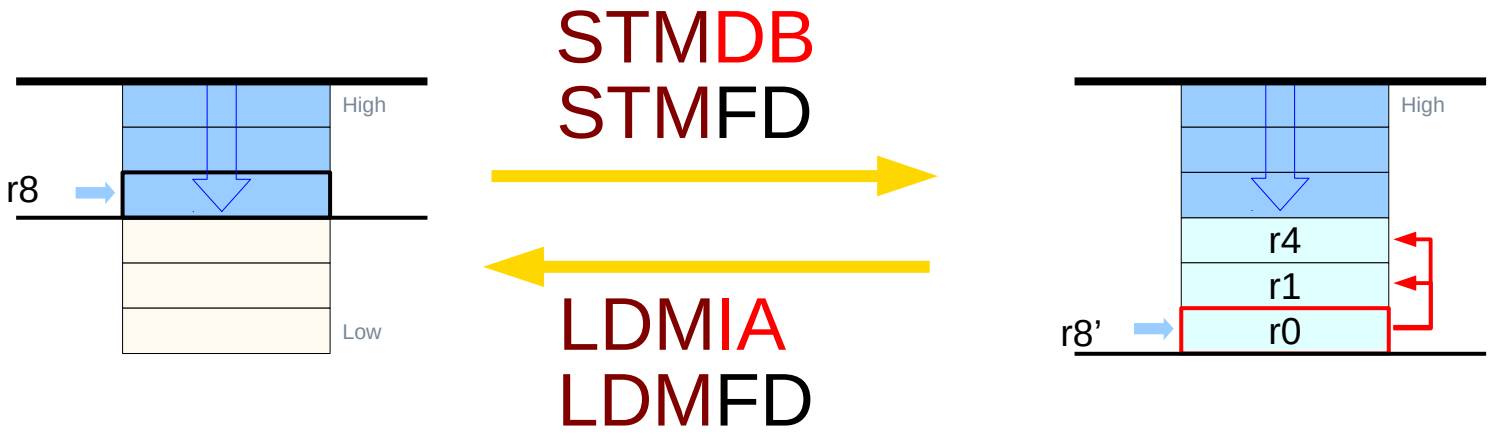
STMIA
STMEA

LDMDB
LDMEA

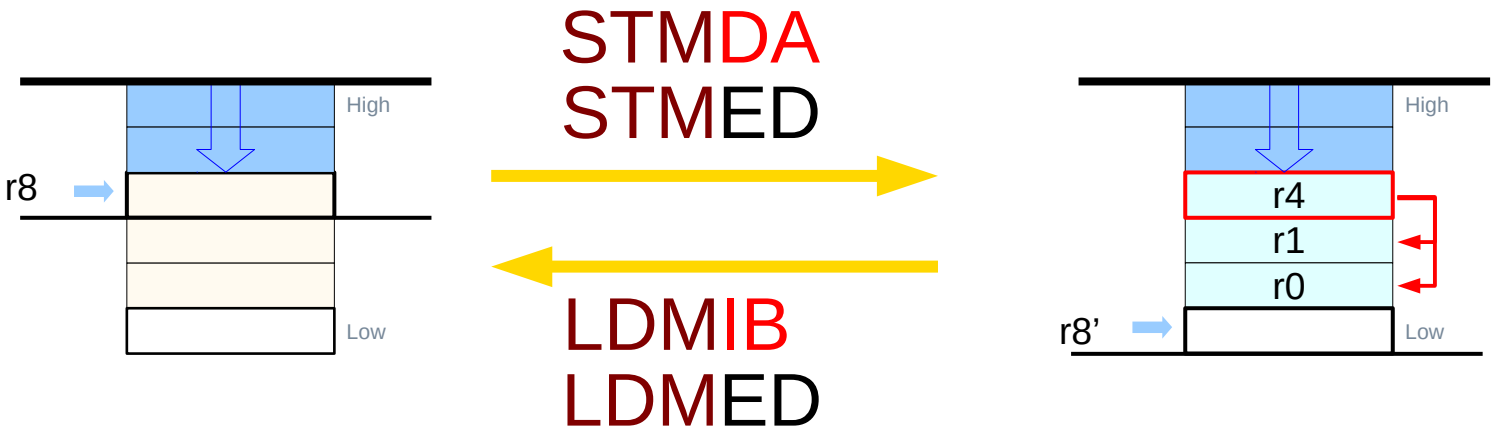


Descending Stack – Equivalent Operations

Full
Descending Stack



Empty
Descending Stack



Multiple Data Transfer – STM (I,D)x(B,A)

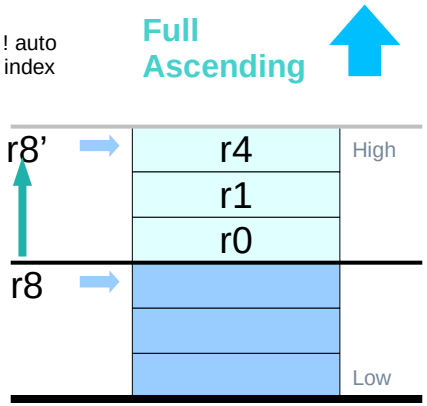
STM Increment Before ST

! auto index

Full Ascending

PUSH

first, increment; then, store




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

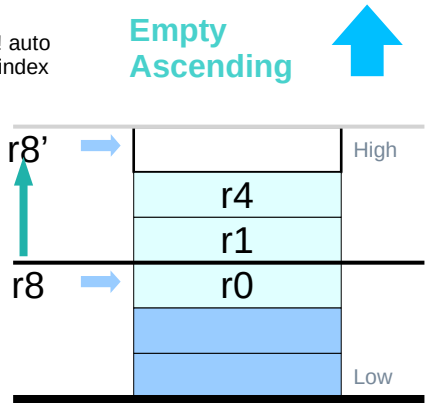
STM Increment After ST

! auto index

Empty Ascending

PUSH

first, store; then, increment




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

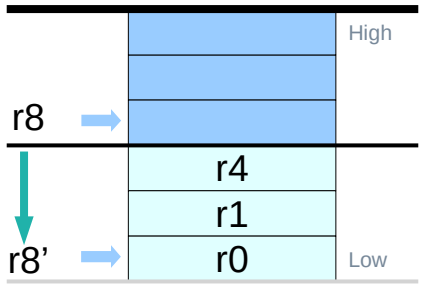
STM Decrement Before ST

! auto index

Full Descending

first, decrement; then, store

PUSH




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

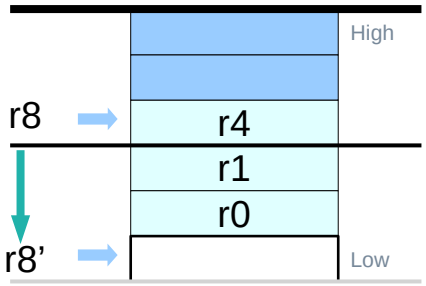
STM Decrement After ST

! auto index

Empty Descending

first, store; then, decrement

PUSH




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

Multiple Data Transfer – LDM (I,D)x(B,A)

LDM

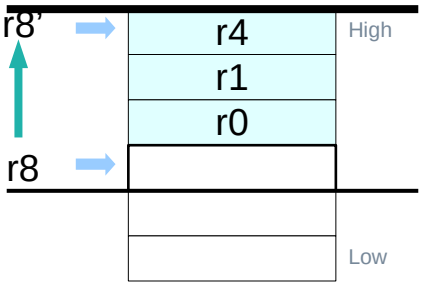
! auto index

Empty Descending

Increment Before LD

POP

first, increment; then, store




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

LDM

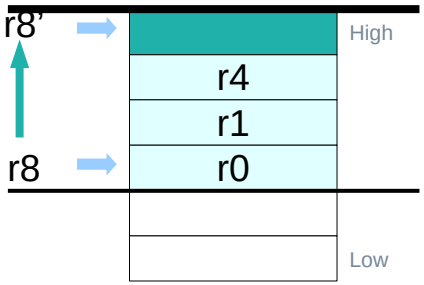
! auto index

Full Descending

Increment After LD

POP

first, store; then, increment




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

LDM

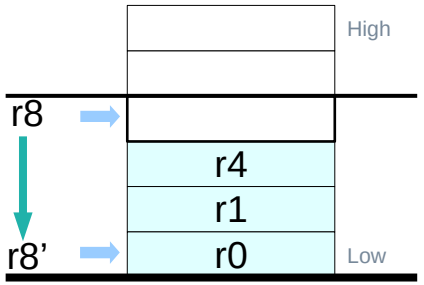
! auto index

Empty Ascending

Decrement Before LD

first, decrement; then, store

POP




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

LDM

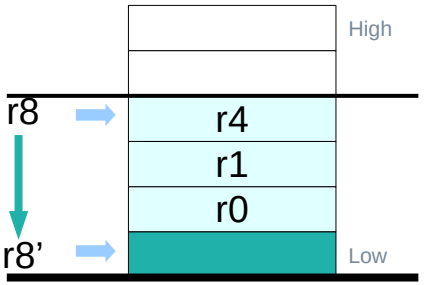
! auto index

Full Ascending

Decrement After LD

first, store; then, decrement

POP




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

Stack and Block Copy Views

		Full	Empty	Full	Empty
		Ascending ↑		Descending ↓	
Increment ↑	Before	STMFA STMIB			LDMED LDMIB
	After		STMEA STMIA	LDMFD LDMIA	
Decrement ↓	Before		LDMEA LDMDB	STMFD STMDB	
	After	LDMFA LDMDA			STMED STMDA

References

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