Block Move Instructions

- The following conventional ARM code demonstrates how to load four registers from consecutive memory locations.

```
ADR r0,DataToGo   ;load r0 with the address of the data area
LDR r1,[r0],#4    ;load r1 with the word pointed at by r0
                  ;and update the pointer
LDR r2,[r0],#4    ;load r2 with the word pointed at by r0
                  ;and update the pointer
LDR r3,[r0],#4    ;load r3 with the word pointed at by r0
                  ;and update the pointer
LDR r5,[r0],#4    ;load r5 with the word pointed at by r0
                  ;and update the pointer
```

- ARM has
  - a ***block move from memory to registers*** instruction, **LDM**, and
  - a ***block move from registers to memory*** instruction, **STM**

  that can copy group of registers from and to memory.

- Both these block move instructions take a suffix to describe ***how*** the data is accessed.

Block Move Instructions

- A block move is easy to understand, because it's simply like
  '*push the contents of these registers to memory*'
  or
  '*pop from the memory the contents of these registers*'.

- Let's start by moving the contents of registers r1, r2, r3, and r5, into sequential memory locations with

```
STMIA r0!,{r1-r3,r5} ;note the syntax of this instruction
                     ;the register list is put
                     ;between curly braces
```

- This instruction copies registers r1 to r3, and r5, into sequential memory locations, using r0 as a pointer with *auto-indexing* (indicated by the **!** suffix).

- The suffix **IA** indicates that index register r0 is *incremented after* the transfer, with data transfer in the order of increasing addresses.

- Although ARM's block move instructions have several variations, *ARM always stores the lowest numbered register first at the lowest memory address*, followed by the next lowest numbered register, and so on.

Block Move Instructions

E:\CengageBook\Z_ARM\MoveMultiple1.uvproj - µVision4

File Edit View Project Flash Debug Peripherals Tools SVCS Window Help

Registers

Register	Value
Current	
R0	0x00000048
R1	0x11111111
R2	0x22222222
R3	0x33333333
R4	0x00000000
R5	0x55555555
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x00000018
CPSR	0x000000D3
SPSR	0x00000000

Project | Registers

MoveMultiple.asm

```
01          AREA MoveMultiple, CODE, READWRITE
02          ENTRY
03 0x00000000 LDR     r1,=0x11111111     ;Let's set up some nice simple values that we can track
04 0x00000004 LDR     r2,=0x22222222
05 0x00000008 LDR     r3,=0x33333333
06 0x0000000C LDR     r5,=0x55555555
07 0x00000010 ADR     r0, Stack          ;let's have a stack to put data on
08 0x00000014 STMIA   r0!,{r1-r3, r5}    ;Now move the data
09 0x00000018 MOV     r0, #0x18          ;angel_SWIreason_ReportException
10 0x0000001C LDR     r1, =0x20026       ;ADP_Stopped_ApplicationExit
11 0x00000020 SVC     #0x123456          ;ARM semihosting (formerly SWI)
12 0x00000024 SPACE   20                 ;leave 20 spaces
13  Stack SPACE   20                     ;leave 20 more spaces
14          END
```

Simulation

Memory 1

Address: 0x0

```
0x0000001C: E5 9F 10 38
0x00000020: EF 12 34 56
0x00000024: 00 00 00 00
0x00000028: 00 00 00 00
0x0000002C: 00 00 00 00
0x00000030: 00 00 00 00
0x00000034: 00 00 00 00
0x00000038: 11 11 11 11
0x0000003C: 22 22 22 22
0x00000040: 33 33 33 33
0x00000044: 55 55 55 55
0x00000048: 00 00 00 00
0x0000004C: 11 11 11 11
0x00000050: 22 22 22 22
0x00000054: 33 33 33 33
0x00000058: 55 55 55 55
```

Call Stack | Locals | Memory 1

r0 contains 0x48 which is the value after the data has been stored.

Registers saved on the stack

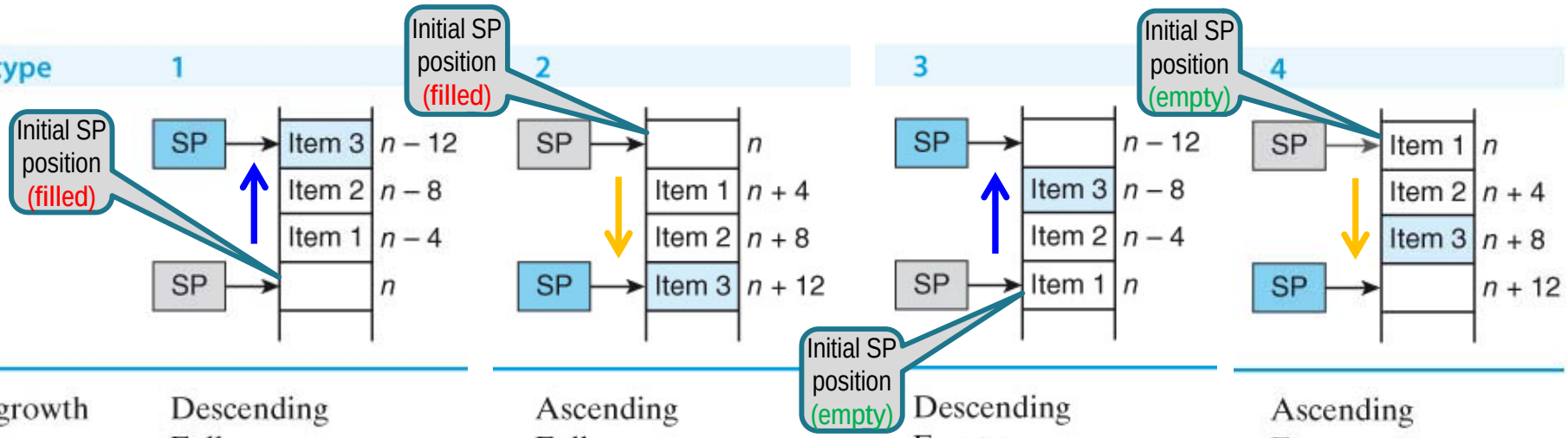
Registers preloaded with data

Registers loaded in the constant pool before execution

- In LDM/STM, the access happens in the order of increasing register numbers,
  - the lowest numbered register occupies the lowest memory address and
  - the highest numbered register occupies the highest memory address

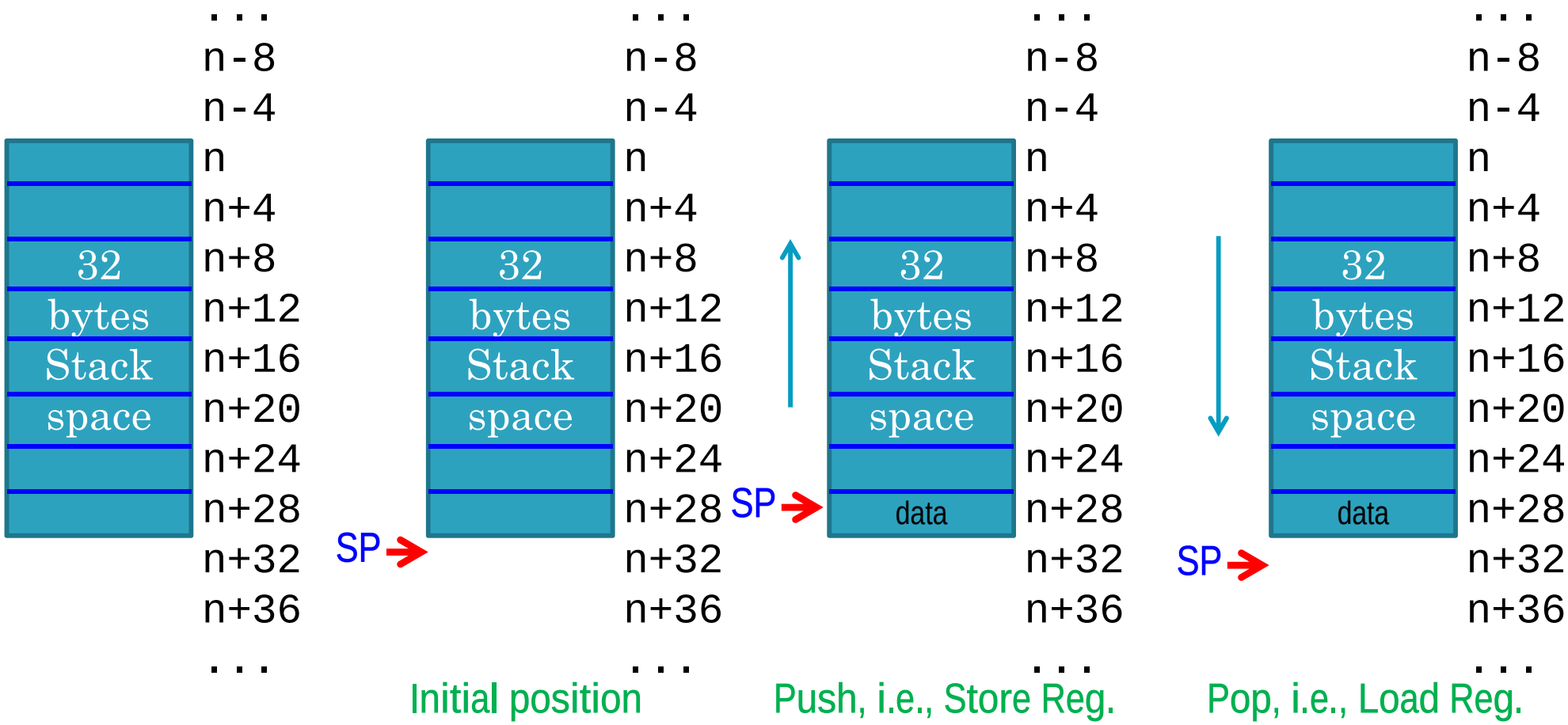
Block Moves and Stack Operations

TABLE 3.5 Stack Types and the ARM Block Move Instruction Suffixes



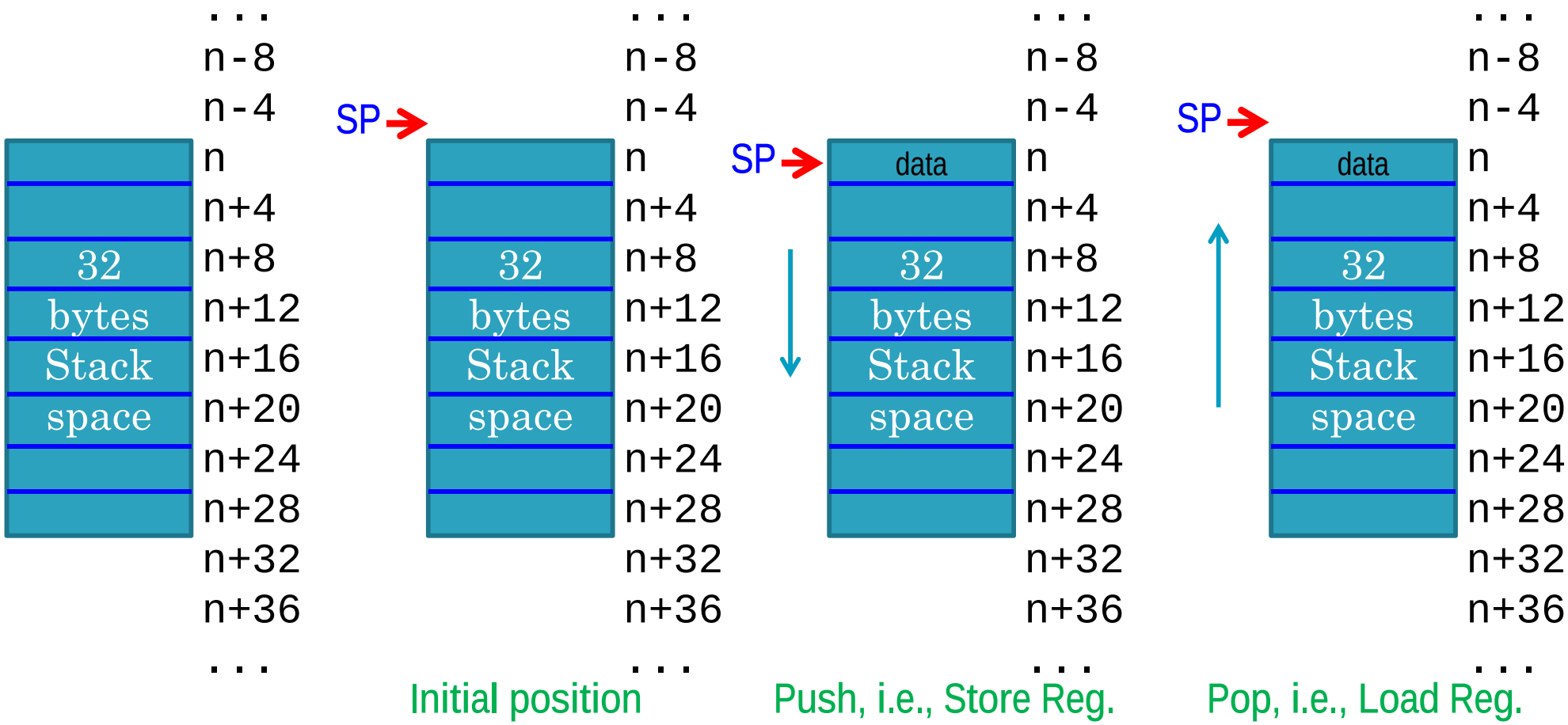
Stack type	1	2	3	4
Stack growth	Descending	Ascending	Descending	Ascending
Class	Full	Full	Empty	Empty
Stack suffix	FD	FA	ED	EA
Load suffix	IA (increment after)	DA (decrement after)	IB (increment before)	DB (decrement before)
Store suffix	DB (decrement before)	IB (increment before)	DA (decrement after)	IA (increment after)

Block Moves and Stack Operations



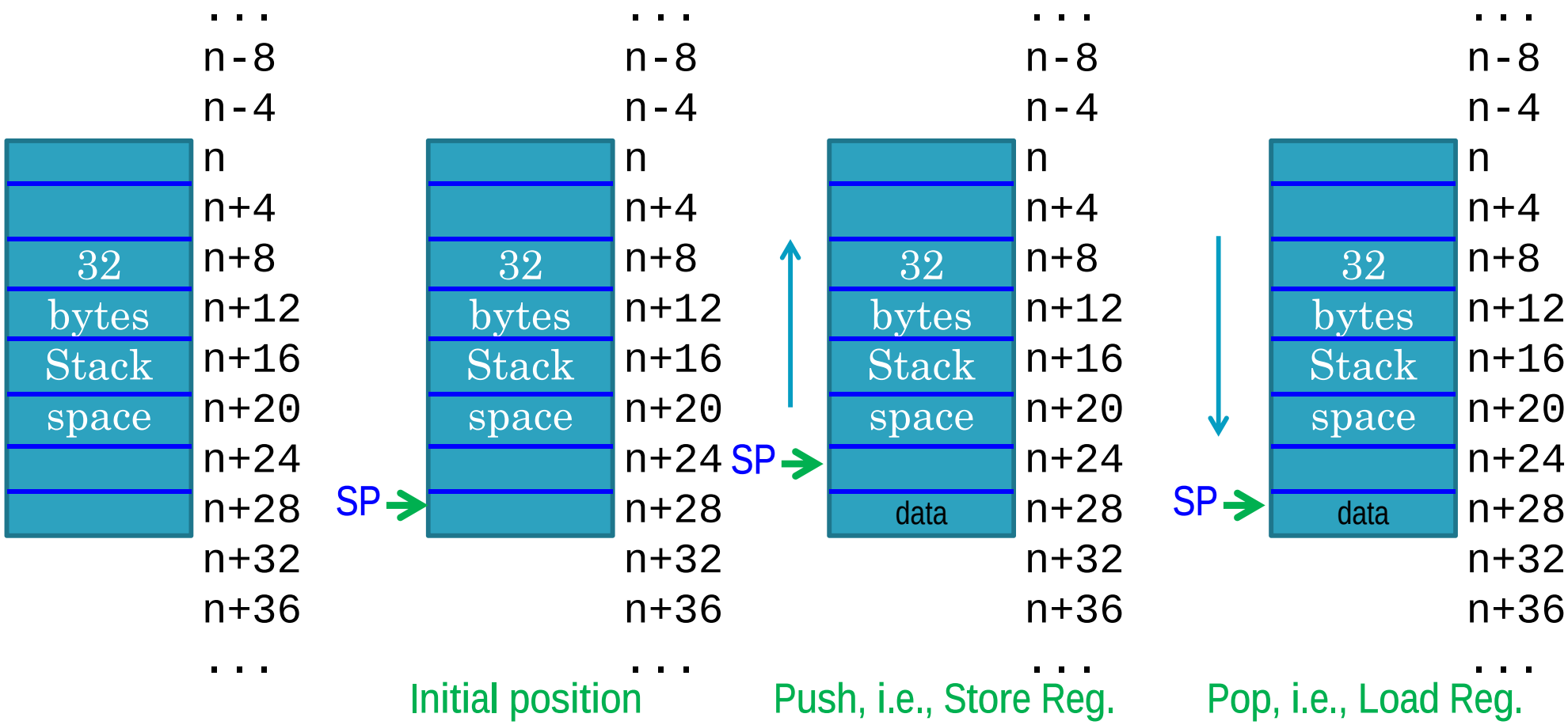
- Stack type: **FD** (i.e., Class=**F**ull and Stack growth=**D**escending) (**STMFD** and **LDMFD**)
  - Empty stack ➔ SP points to just after the stack space
  - Pushing on the stack ➔ SP to be **D**ecremented **B**efore (**STMDB**)
  - Popping off the stack ➔ SP to be **I**ncremented **A**fter (**LDMIA**)

Block Moves and Stack Operations



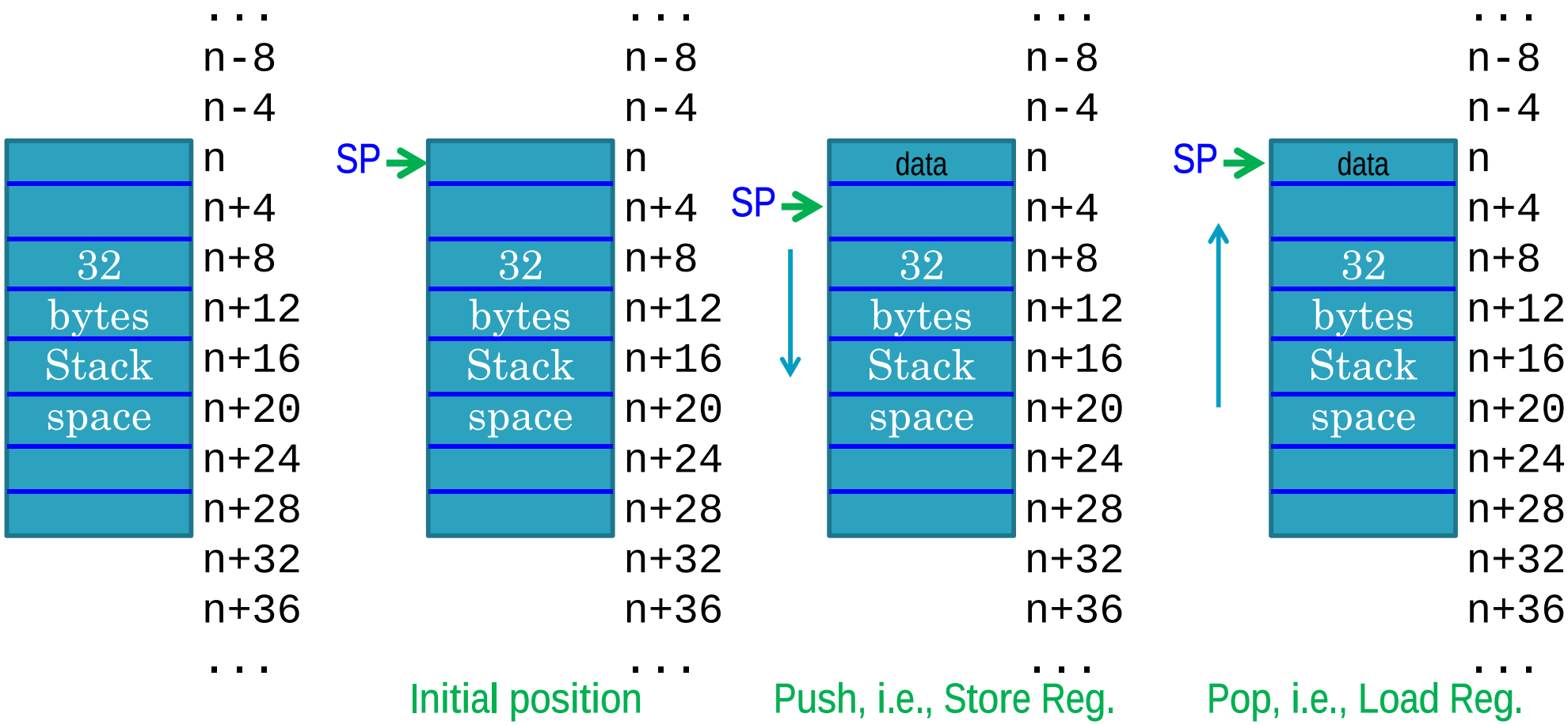
- Stack type: **FA** (i.e., Class=**F**ull and Stack growth=**A**scending) (**STMFA** and **LDMFA**)
  - Empty stack ➔ SP points to just before the stack space
  - Pushing on the stack ➔ SP to be **I**ncremented **B**efore (**STMIB**)
  - Popping off the stack ➔ SP to be **D**ecremented **A**fter (**LDMDA**)

Block Moves and Stack Operations



- Stack type: **ED** (i.e., Class=**E**mpty and Stack growth=**D**escending) (**STMED** and **LDMED**)
  - Empty stack ➔ SP points to the last memory word in the stack
  - Pushing on the stack ➔ SP to be **D**ecremented **A**fter (**STMDA**)
  - Popping off the stack ➔ SP to be **I**ncremented **B**efore (**LDMIB**)

Block Moves and Stack Operations



- Stack type: **EA** (i.e., Class=**E**mpty and Stack growth=**A**scending) (**STMEA** and **LDMEA**)
  - Empty stack ➔ SP points to the first memory word in the stack
  - Pushing on the stack ➔ SP to be **I**ncremented **A**fter (**STMIA**)
  - Popping off the stack ➔ SP to be **D**ecremented **B**efore (**LDMDB**)

Block Moves and Stack Operations

- ARM has *two* ways of describing stacks, which can be a little confusing at first.
- A stack operation can be described either by
  - ***what*** it does
    - *FD* *Full Descending* ← *most popular stack*
    - *FA* *Full Ascending*
    - *ED* *Empty Descending*
    - *EA* *Empty Ascending*
  - ***how*** it does
    - ***DB*** ***D****ecremented* ***B****efore*
    - ***DA*** ***D****ecremented* ***A****fter*
    - ***I B*** ***I****ncremented* ***B****efore*
    - ***I A*** ***I****ncremented* ***A****fter*

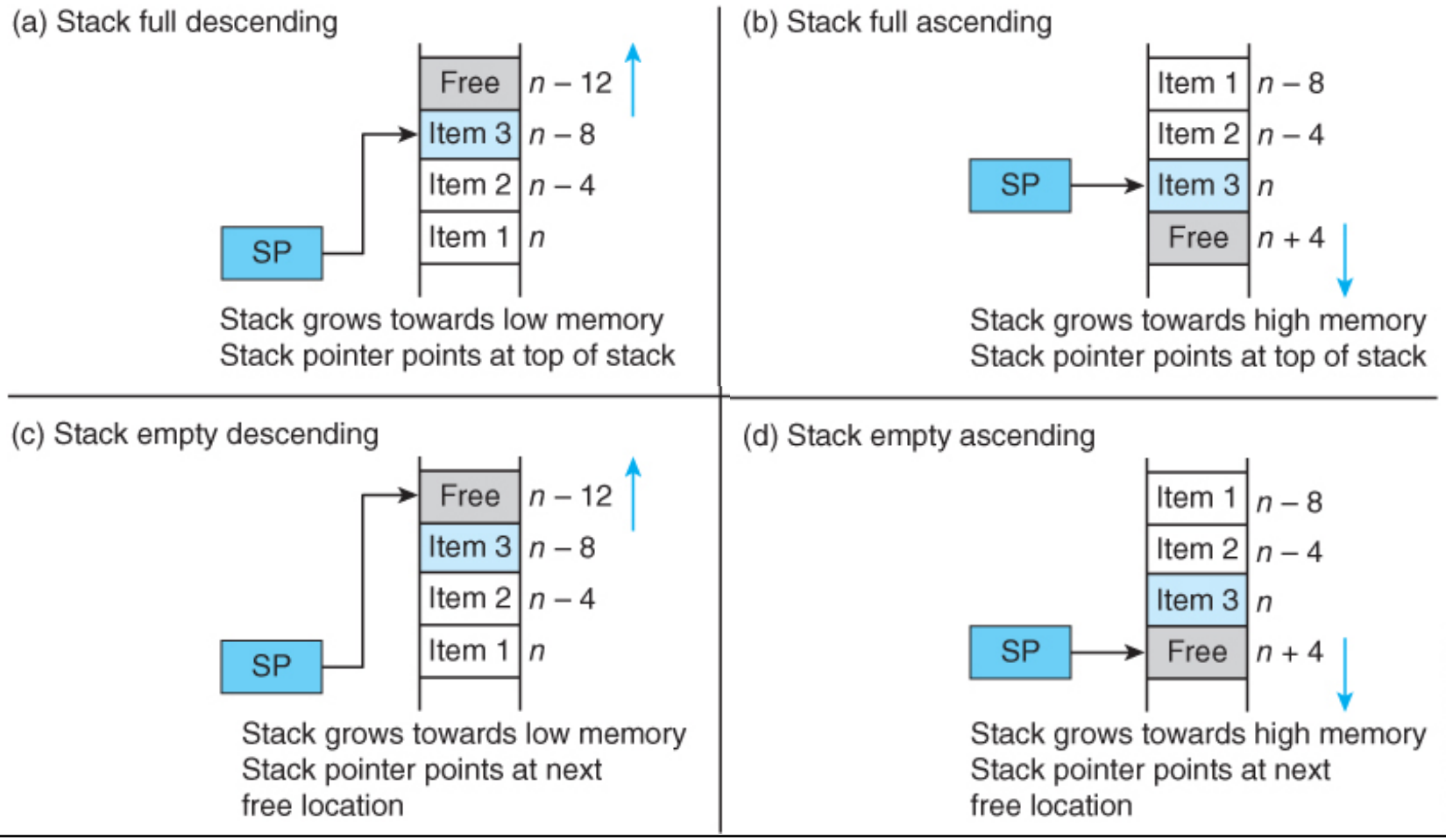
For example,

- We can write `STMFD sp!,{r0,r1}` when pushing r0 and r1 on the stack,
  - Also can be written as `STMDB sp!,{r0,r1}`
- We can write `LDMFD sp!,{r0,r1}` when popping r0 and r1 off the stack.
  - Also can be written as `LDMIA sp!,{r0,r1}`

Block Moves and Stack Operations

- The ARM's literature uses four terms to describe stacks:
  - a) FD *full descending* Figure 3.52a
  - b) FA *full ascending* Figure 3.52b
  - c) ED *empty descending* Figure 3.52c
  - d) EA *empty ascending* Figure 3.52d

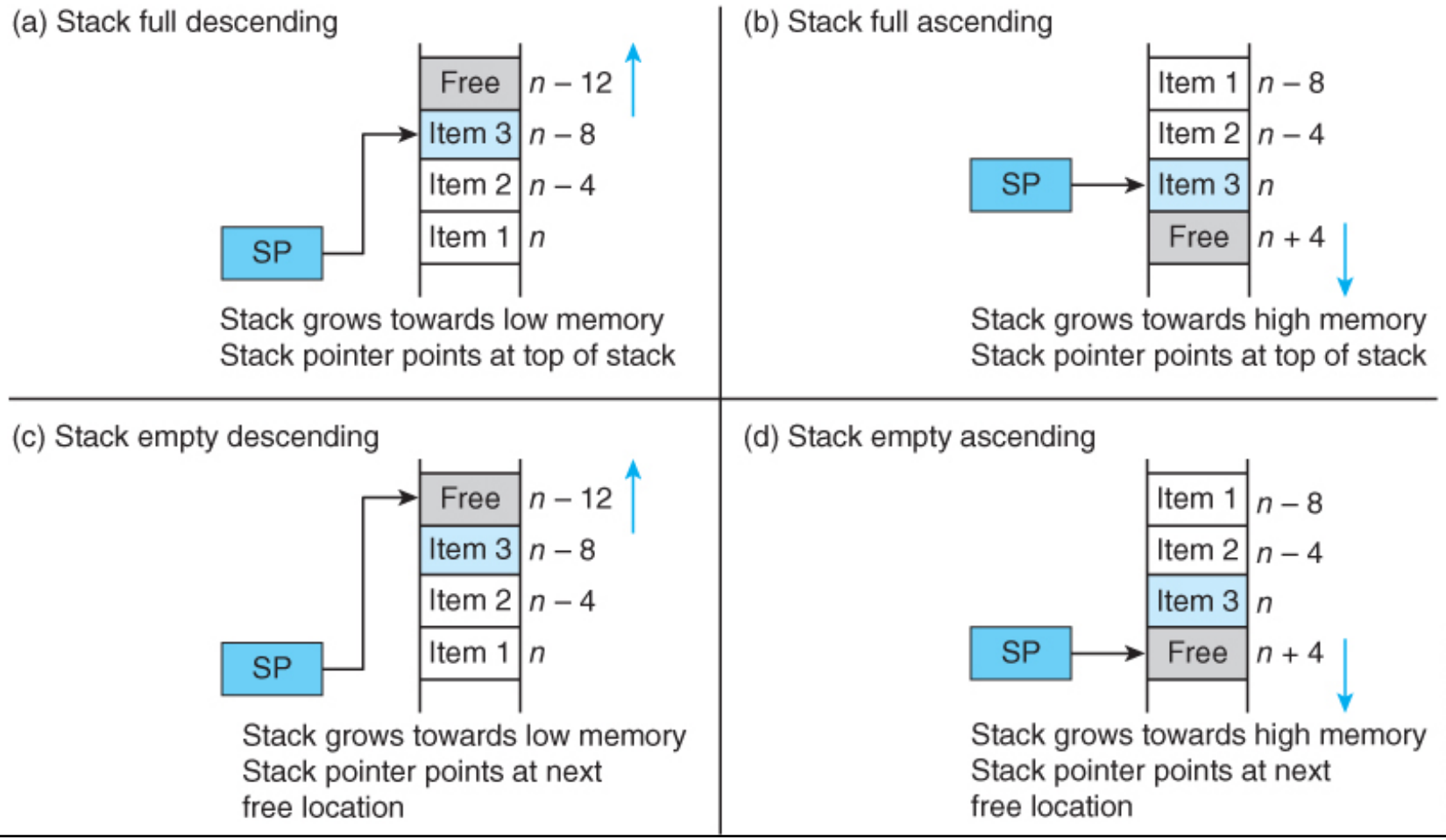
FIGURE 3.59 ARM's four stack modes



Block Moves and Stack Operations

- A stack is described as *full* if the stack pointer *points to the top element* of the stack.
- If the stack pointer *points to the next free* element in the stack, then the stack is called *empty*.

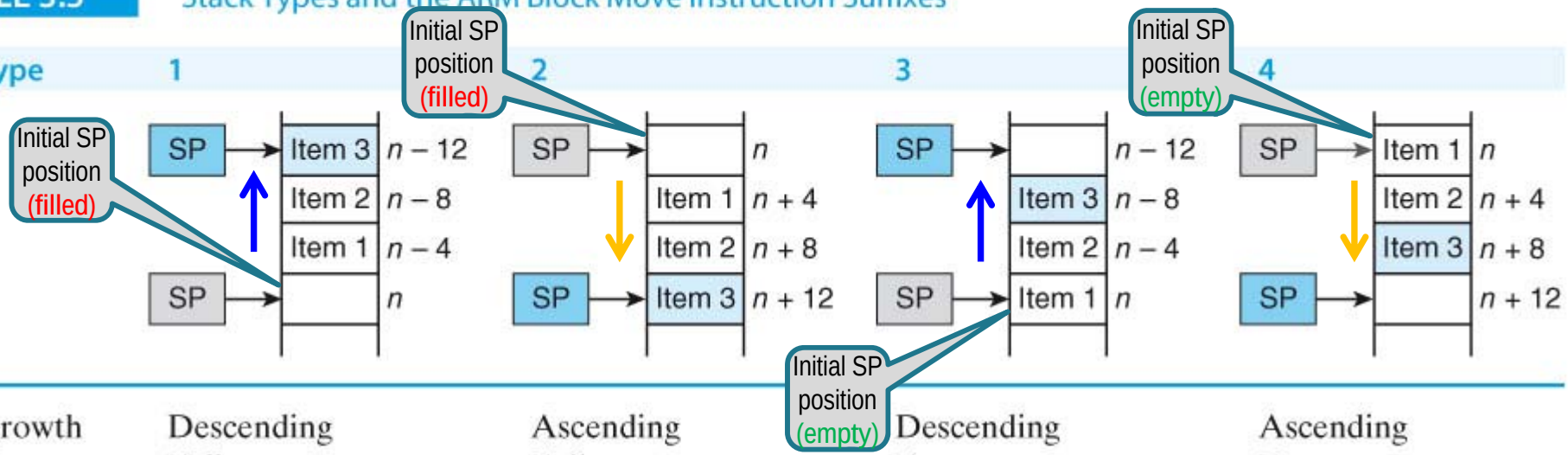
FIGURE 3.59 ARM's four stack modes



Block Moves and Stack Operations

FD, FA, ED, and **EA** are ***pseudo*** notation.
There are translated to the **IA, DB, DA,** and **IB** notation.

TABLE 3.5 Stack Types and the ARM Block Move Instruction Suffixes



Stack growth	Descending	Ascending	Descending	Ascending
Class	Full	Full	Empty	Empty
Stack suffix	FD	FA	ED	EA
Load suffix	IA (increment after)	DA (decrement after)	IB (increment before)	DB (decrement before)
Store suffix	DB (decrement before)	IB (increment before)	DA (decrement after)	IA (increment after)

Block Moves and Stack Operations

- ARM's block move instruction is useful because it supports four possible stack modes.
- The differences among these modes are
  - the *direction* in which the stack grows
    - up (i.e., *ascending*), *or*
    - down (i.e., *descending*)
  - whether the *stack pointer points at*
    - the item currently at the *top of the stack* or
    - the *next free item* on the stack.

Bit#23 (Up/down)

Bit#24 (Pre-post)

- ARM uses the terms *ascending* and *descending* to describe the growth of the stack toward *higher* or *lowers* addresses, respectively.

- **CISC** processors with hardware stack support generally provide ***only one*** fixed stack mode.

Applications of Block Move Instructions

- One of the most important applications of the ARM's block move instructions is in
  - saving registers on entering a subroutine and
  - restoring registers before returning from a subroutine.
- Consider the following ARM code:

```
     BL    test                    ;call test, save return
                                   ;address in r14
     ...
test STMFD r13!,{r0-r4,r10}        ;subroutine test, save working
                                   ;registers
     . body of code
     .
     LDMFD r13!,{r0-r4,r10}        ;subroutine completes,
                                   ;restore the registers
     MOV   pc,r14                  ;copy the return address in
                                   ;r14 to the PC
```

Applications of Block Move Instructions

- If you are using a block move to restore registers from the stack, you can also include the program counter.

We can write:

```
test STMFD r13!,{r0-r4,r10,r14} ;save working registers
                                ;and return address in r14
        :
     LDMFD r13!,{r0-r4,r10,r15} ;restore working registers
                                ;and put r14 in the PC
```

- At the beginning of the subroutine we push the ***link register r14*** containing the return address onto the stack, and then at the end we pull the saved registers, including the value of the return address which is placed in the ***PC***, to effect the return.
  - By doing so, we reduced the size of this code by one instruction

Applications of Block Move Instructions

- The block move provides a convenient means of copying data between memory regions.
- In the next example we copy 256 words (1024 bytes) from Table 1 to Table 2.

```
     ADR     r0,Table1     ;r0 points to source
                           ;(note pseudo-op ADR)
     ADR     r1,Table2     ;r1 points to the destination
     MOV     r2,#32        ;32 blocks of 8 = 256 words to move
Loop LDMFD   r0!,{r3-r10}  ;REPEAT Load 8 registers (r3 to r10)
     STMFD   r1!,{r3-r10}  ;store the registers at
                           ;their destination
     SUBS    r2,r2,#1      ;decrement loop counter
     BNE     Loop          ;UNTIL all 32 blocks of
                           ;8 registers moved
```

Is it right to use LDMFD and STMFD?

- The two block move instructions above allow us to move eight registers (i.e., 32 bytes) at once.

Applications of Block Move Instructions

- The block move provides a convenient means of copying data between memory regions.
- In the next example we copy 256 words (1024 bytes) from Table 1 to Table 2.

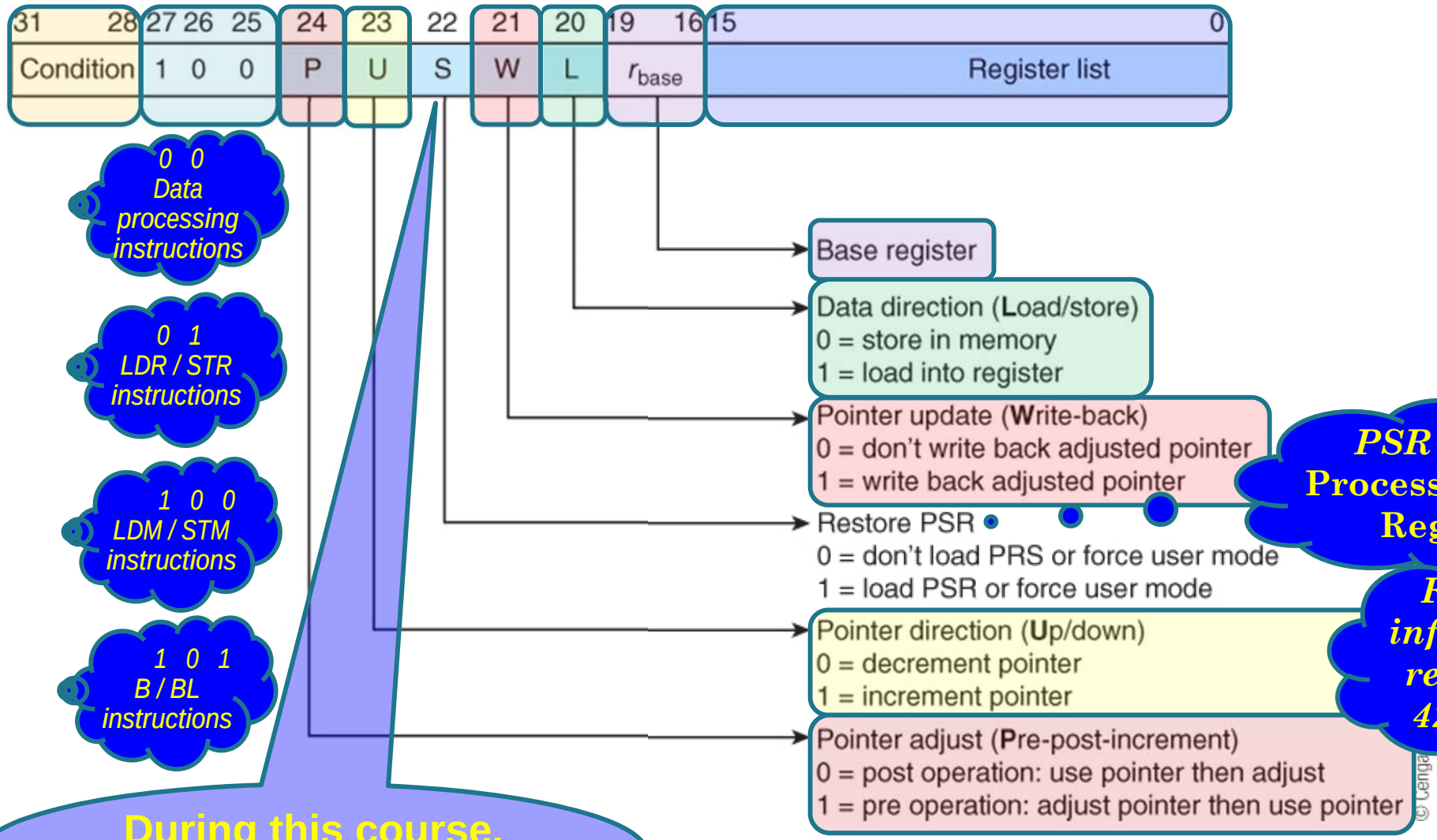
```
     ADR     r0,Table1     ;r0 points to source
                           ;(note pseudo-op ADR)
     ADR     r1,Table2     ;r1 points to the destination
     MOV     r2,#32        ;32 blocks of 8 = 256 words to move
Loop LDMIA   r0!,{r3-r10}  ;REPEAT Load 8 registers (r3 to r10)
     STMIA   r1!,{r3-r10}  ;store the registers at
                           ;their destination
     SUBS    r2,r2,#1      ;decrement loop counter
     BNE     Loop          ;UNTIL all 32 blocks of
                           ;8 registers moved
```

LDMIA and **STMIA**,
not LDRFD and **STRFD**
Not correct in the book page 220

- The two block move instructions above allow us to move eight registers (i.e., 32 bytes) at once.

Block Move Instructions Encoding/Decoding

FIGURE 3.58 Encoding ARM's block move instructions



PSR means Processor Status Register

For more information, read slides 42 and 43.

During this course, it will always be 0

Block Move Instructions Encoding Example

ARM Instruction: STMFD **r13!,{r0-r4,r10}**

Condition = 1110 (always – unconditional)

P = 1 (DB: adjust pointer then use pointer)

U = 0 (DB: decrement)

S = 0 (user mode)

W = 1 (write-back adjusted pointer)

L = 0 (store)

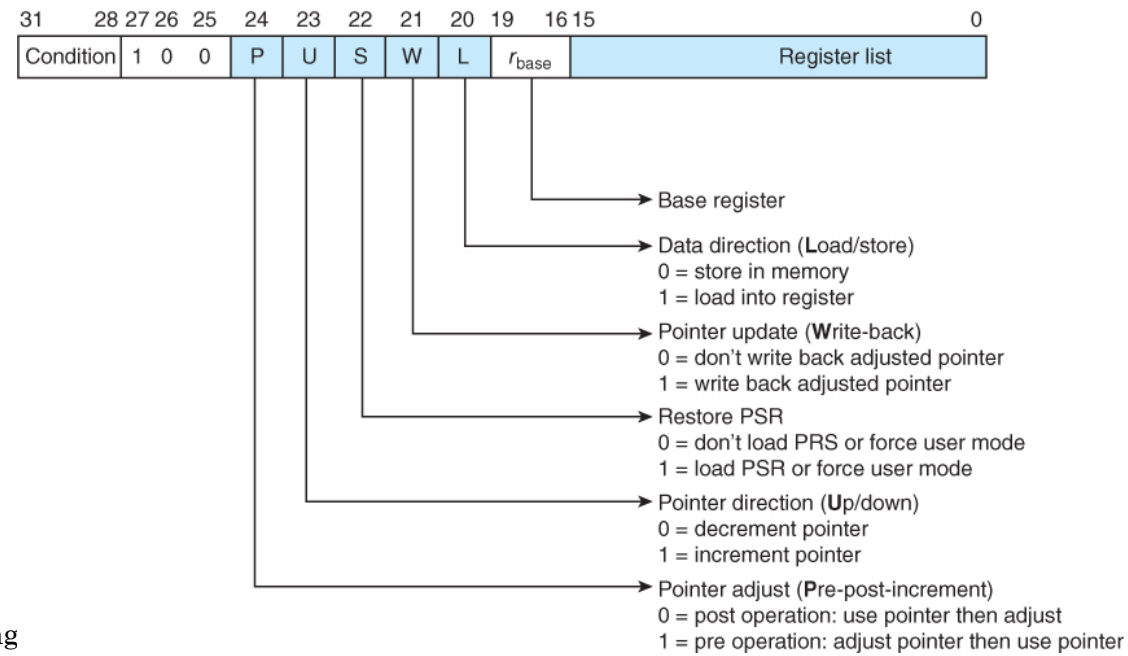
r_{base} = 1101 (r13)

Register list (r15, r14, …., r2, r1, r0) = 0000 0100 0001 1111

1110 1001 0010 1101 0000 0100 0001 1111

0xE92D041F

FIGURE 3.58 Encoding ARM's block move instructions



Block Move Instructions Encoding Example

ARM Instruction: `LDMFD r13!,{r0-r4,r10}`

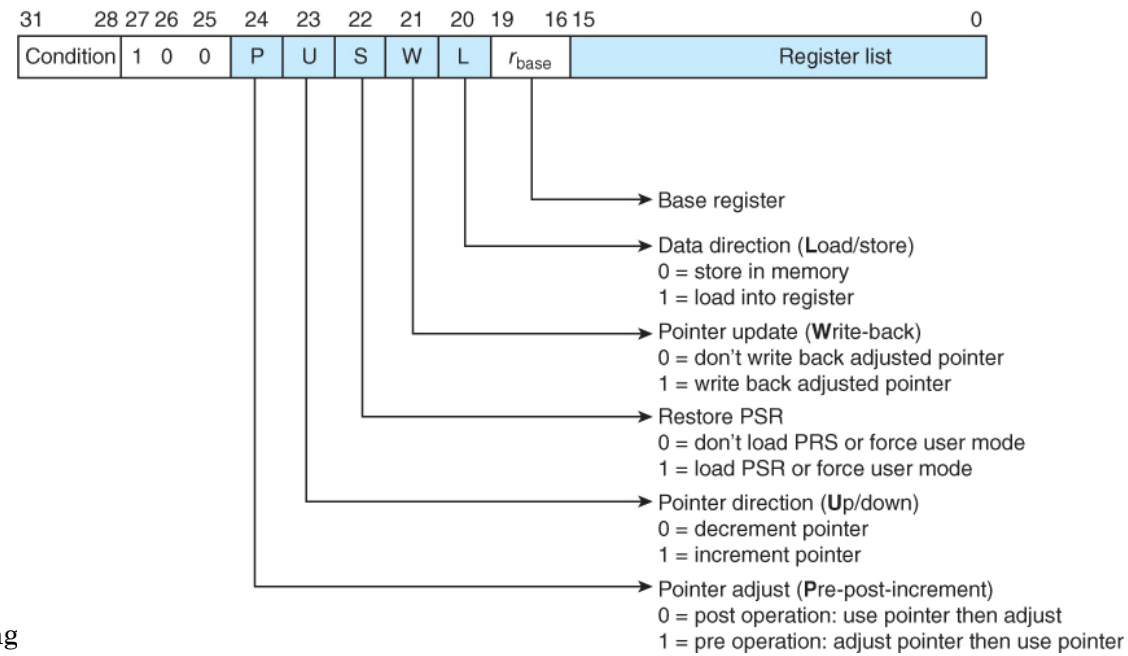
Condition = 1110 (always – unconditional)

- P = 0 (IA: use pointer then adjust)
- U = 1 (IA: increment)
- S = 0 (user mode)
- W = 1 (write-back adjusted pointer)
- L = 1 (load)
- r_{base} = 1101 (r13)
- Register list (r15, r14, …., r2, r1, r0) = 0000 0100 0001 1111

1110 1000 1011 1101 0000 0100 0001 1111

0xE8BD041F

FIGURE 3.58 Encoding ARM's block move instructions



Block Move Instructions Decoding Example

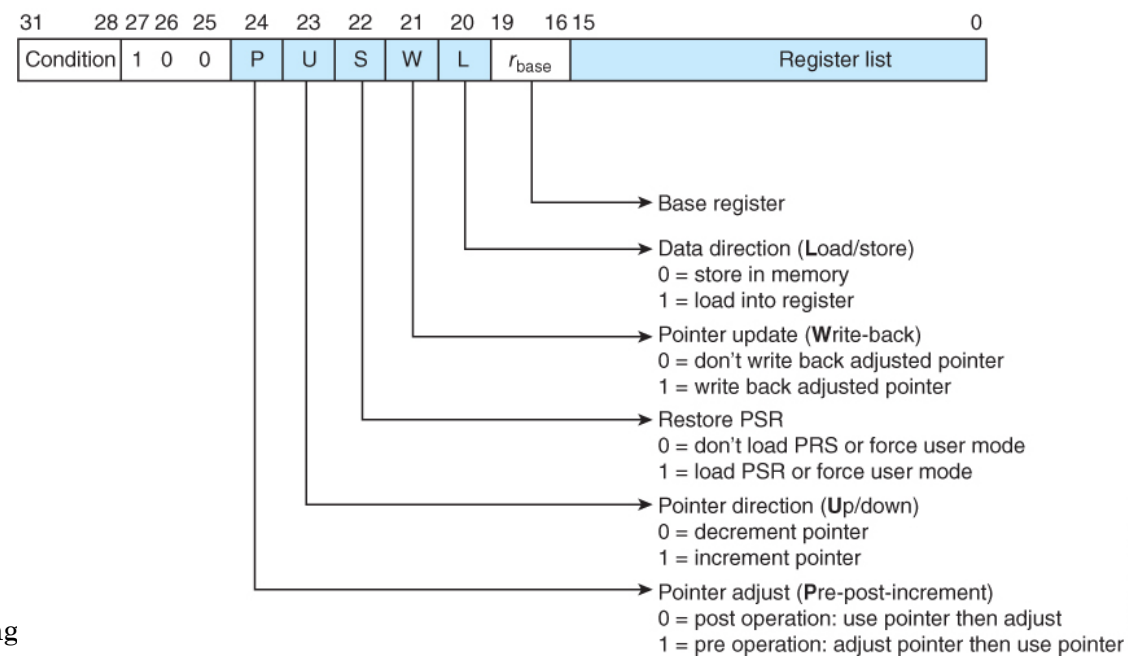
Decode the ARM machine language **0x08855555**

```
    0000 1000 1000 0101 0101 0101 0101 0101
Condition = 0000 (EQ)
    P = 0 (IA: use pointer then adjust)
    U = 1 (IA: increment)
    S = 0 (user mode)
    W = 0 (do not write-back adjusted pointer)
    L = 0 (store)
    r_base = 0101 (r5)
    Register list (r15, r14, …., r2, r1, r0) = 0101 0101 0101 0101
```

ARM Instruction: `STMEQIA r5,{r0,r2,r4,r6,r8,r10,r12,r14}`

It can also be
STMIAEQ
STMEQEA
STMEAEQ

FIGURE 3.58 Encoding ARM's block move instructions



Block Move Instructions Decoding Example

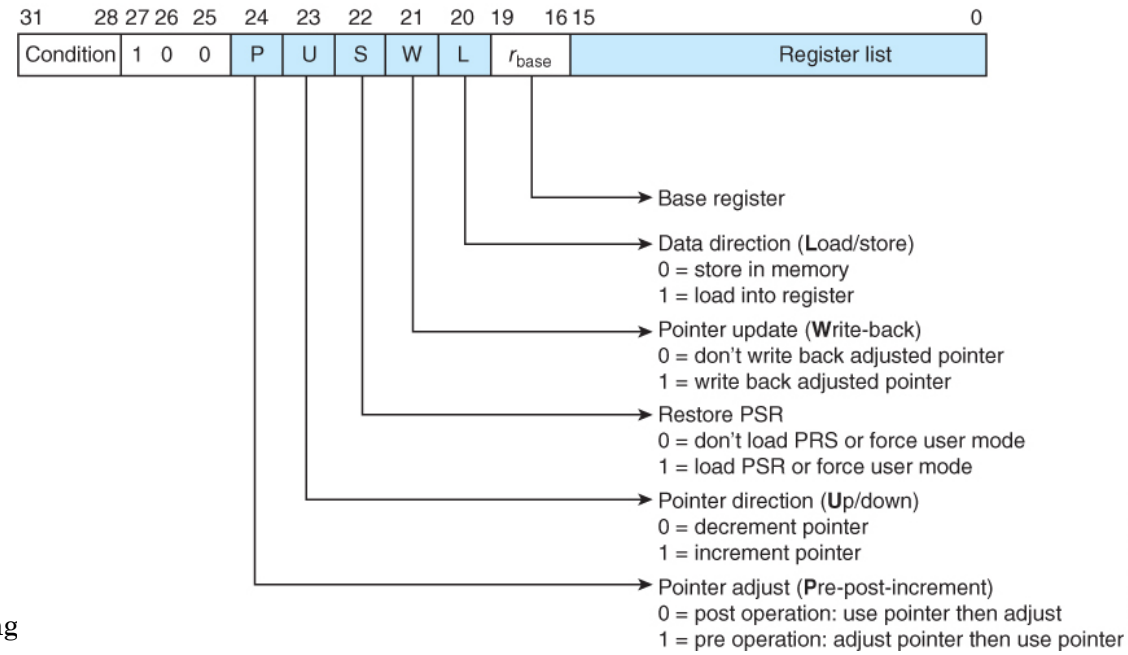
Decode the ARM machine language **0x99922222**

```
    1001 1001 1001 0010 0010 0010 0010 0010
Condition = 1001 (LS)
    P = 1 (IB: adjust pointer then use pointer)
    U = 1 (IB: increment)
    S = 0 (user mode)
    W = 0 (do not write-back adjusted pointer)
    L = 1 (load)
    r_base = 0010 (r2)
    Register list (r15, r14, ...., r2, r1, r0) = 0010 0010 0010 0010
```

ARM Instruction: `LDMLSIB r2,{r1,r5,r9,r13}`

It can also be
LDMIBLS
LDMLSED
LDMEDLS

FIGURE 3.58 Encoding ARM's block move instructions



ARM Assembly Instructions Summary

ADC{cond}{S} {Rd,}Rn,Op2	Add with carry	Rd ← Rn + Op2 + Carry
ADD{cond}{S} {Rd,}Rn,Op2	Add	Rd ← Rn + Op2
MLA{cond}{S} Rd, Rm,Rs,Rn	Multiply Accumulate	Rd ← (Rm × Rs) + Rn
MUL{cond}{S} Rd, Rm,Rs	Multiply	Rd ← Rm × Rs
MOV{cond}{S} Rd,Op2	Move register or constant	Rd ← Op2
NEG{cond}{S} Rd,Rn	Negate the value in a register	Rd ← - Rn
RSB{cond}{S}{ Rd,}Rn,Op2	Reverse Subtract	Rd ← Op2 - Rn
RSC{cond}{S} {Rd,}Rn,Op2	Reverse Subtract with Carry	Rd ← Op2 - Rn - 1 + Carry
SBC{cond}{S} {Rd,}Rn,Op2	Subtract with Carry	Rd ← Rn - Op2 - 1 + Carry
SUB{cond}{S} {Rd,}Rn,Op2	Subtract	Rd ← Rn - Op2

ARM Assembly Instructions Summary

AND{cond}{S} {Rd,}Rn,Op2	AND	Rd ← Rn *AND* Op2
BIC{cond}{S} {Rd,}Rn,Op2	Bit Clear	Rd ← Rn *AND NOT* Op2
ORR{cond}{S} {Rd,}Rn,Op2	OR	Rd ← Rn *OR* Op2
EOR{cond}{S} {Rd,}Rn,Op2	Exclusive OR	Rd ← Rn ⊕ Op2
MVN{cond}{S} Rd,Op2	Move not	Rd ← 0xFFFFFFFF ⊕ Op2

ARM Assembly Instructions Summary

CMN{cond} Rn,Op2	Compare Negative	CPSR flags ← Rn + Op2
CMP{cond} Rn,Op2	Compare	CPSR flags ← Rn - Op2
TEQ{cond} Rn,Op2	Test bitwise equality	CPSR flags ← Rn ⊕ Op2
TST{cond} Rn,Op2	Test bits	CPSR flags ← Rn *AND* Op

ARM Assembly Instructions Summary

B{cond} address	Branch	R15 ← address
BL{cond} address	Branch with Link	R14 ← R15, R15 ← address

ARM Assembly Instructions Summary

ADR{cond}Rd,label	Load address	Rd ← The address of the label
STR{cond}{B} Rd,address	Store register to memory	[address] ← Rd
LDR{cond}{B} Rd,address	Load register from memory	Rd ← [address]
LDR{cond} Rd,=expr	Load a 32-bit immediate value	Rd ← expr
LDR{cond} Rd,=label	Load a 32-bit address	Rd ← The address of the label

ARM Assembly Instructions Summary

LDM{cond}{IA|IB|DA|DB}{cond} Rn{!},reglist{^}	Load Multiple registers/Stack pop
LDM{cond}{FD|FA|ED|EA}{cond} Rn{!},reglist{^}	Load Multiple registers/Stack pop
STM{cond}{IA|IB|DA|DB}}{cond} Rn{!},reglist{^}	Store Multiple registers/Stack push
STM{cond}{FD|FA|ED|EA}}{cond} Rn{!},reglist{^}	Store Multiple registers/Stack push

ARM Assembly Instructions Summary

ADC{cond}{S} {Rd,}Rn,Op2	Add with carry	Rd ← Rn + Op2 + Carry
ADD{cond}{S} {Rd,}Rn,Op2	Add	Rd ← Rn + Op2
AND{cond}{S} {Rd,}Rn,Op2	AND	Rd ← Rn *AND* Op2
ADR{cond}Rd,label	Load address	Rd ← The address of the label
B{cond} address	Branch	R15 ← address
BIC{cond}{S} {Rd,}Rn,Op2	Bit Clear	Rd ← Rn *AND NOT* Op2
BL{cond} address	Branch with Link	R14 ← R15, R15 ← address
CMN{cond} Rn,Op2	Compare Negative	CPSR flags ← Rn + Op2
CMP{cond} Rn,Op2	Compare	CPSR flags ← Rn - Op2
EOR{cond}{S} {Rd,}Rn,Op2	Exclusive OR	Rd ← Rn ⊕ Op2
LDM{cond}{IA|IB|DA|DB}{cond} Rn{!},reglist{^}		Load Multiple registers/Stack pop
LDM{cond}{FD|FA|ED|EA}{cond} Rn{!},reglist{^}		Load Multiple registers/Stack pop
LDR{cond}{B} Rd,address	Load register from memory	Rd ← [address]
LDR{cond} Rd,=expr	Load a 32-bit immediate value	Rd ← expr
LDR{cond} Rd,=label	Load a 32-bit address	Rd ← The address of the label
MLA{cond}{S} Rd, Rm,Rs,Rn	Multiply Accumulate	Rd ← (Rm × Rs) + Rn
MOV{cond}{S} Rd,Op2	Move register or constant	Rd ← Op2
MUL{cond}{S} Rd, Rm,Rs	Multiply	Rd ← Rm × Rs
MVN{cond}{S} Rd,Op2	Move not	Rd ← 0xFFFFFFFF ⊕ Op2
NEG{cond}{S} Rd,Rn	Negate the value in a register	Rd ← - Rn
NOP	No operation	No operation
ORR{cond}{S} {Rd,}Rn,Op2	OR	Rd ← Rn *OR* Op2
RSB{cond}{S}{ Rd,}Rn,Op2	Reverse Subtract	Rd ← Op2 - Rn
RSC{cond}{S} {Rd,}Rn,Op2	Reverse Subtract with Carry	Rd ← Op2 - Rn - 1 + Carry
SBC{cond}{S} {Rd,}Rn,Op2	Subtract with Carry	Rd ← Rn - Op2 - 1 + Carry
STM{cond}{IA|IB|DA|DB}}{cond} Rn{!},reglist{^}		Store Multiple registers/Stack push
STM{cond}{FD|FA|ED|EA}}{cond} Rn{!},reglist{^}		Store Multiple registers/Stack push
STR{cond}{B} Rd,address	Store register to memory	[address] ← Rd
SUB{cond}{S} {Rd,}Rn,Op2	Subtract	Rd ← Rn - Op2
TEQ{cond} Rn,Op2	Test bitwise equality	CPSR flags ← Rn ⊕ Op2
TST{cond} Rn,Op2	Test bits	CPSR flags ← Rn *AND* Op2

{S} ➔ Update condition flags if S present
{cond} ➔ *(to be omitted for unconditional execution)*
Refer to the table below for the meaning of the {cond} field.

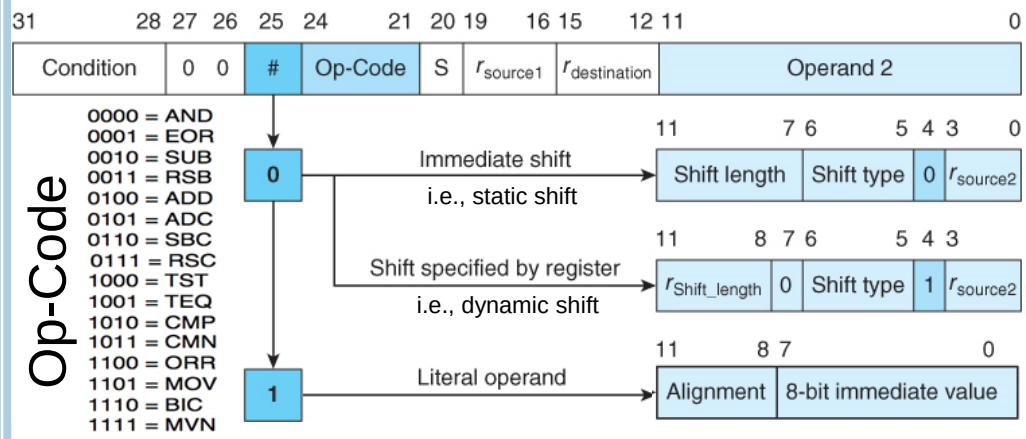
ARM Assembly Instructions Summary

Meaning of {condition} field

Encoding	Mnemonic	Branch on Flag Status	Execute on Condition
0000	EQ	Z set	Equal (i.e., zero)
0001	NE	Z clear	Not equal (i.e., not zero)
0010	CS	C set	Unsigned higher or same
0011	CC	C clear	Unsigned lower
0100	MI	N set	Negative
0101	PL	N clear	Positive or zero
0110	VS	V set	Overflow
0111	VC	V clear	No overflow
1000	HI	C set and Z clear	Unsigned higher
1001	LS	C clear or Z set	Unsigned lower or same
1010	GE	N set and V set, or N clear and V clear	Greater or equal
1011	LT	N set and V clear, or N clear and V set	Less than
1100	GT	Z clear and N set and V set, or Z clear and N clear and V clear	Greater than
1101	LE	Z set, or N set and V clear, or N clear and V set	Less than or equal
1110	AL		Always (default)
1111	NV		Never (reserved)

Instruction Encoding Formats

FIGURE 3.26 Encoding the ARM's data processing instructions



Shift type
00 = logical left
01 = logical right
10 = arithmetic right
11 = rotate right

FIGURE 3.41 Encoding ARM's branch and branch-with-link instructions

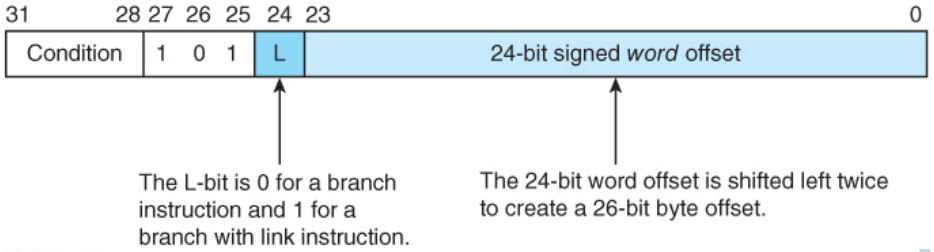


FIGURE 3.39 Format of ARM's load and store instructions

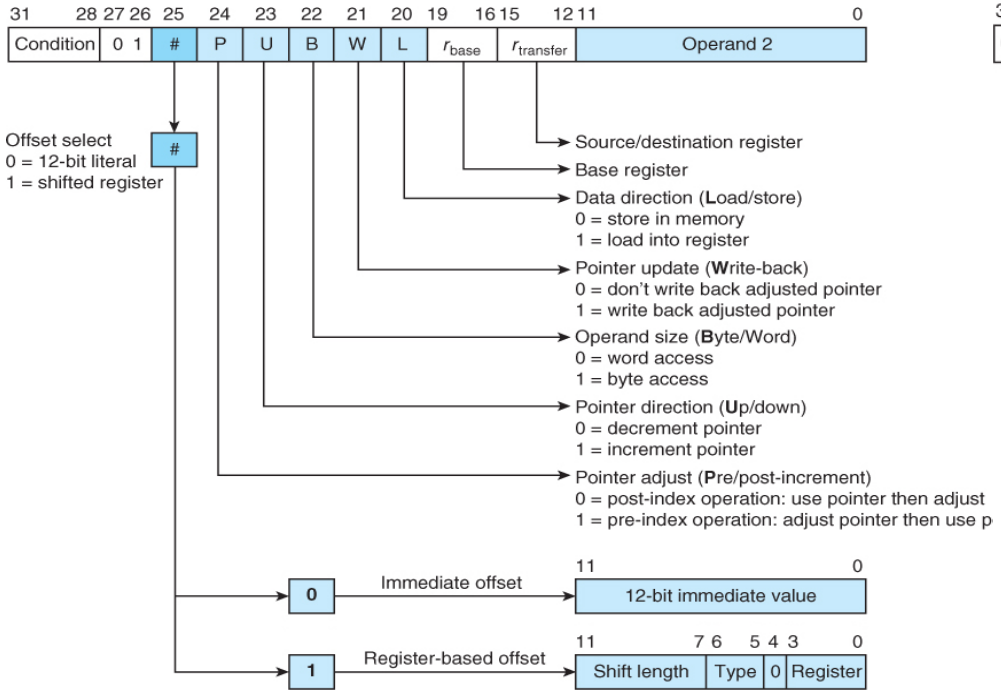
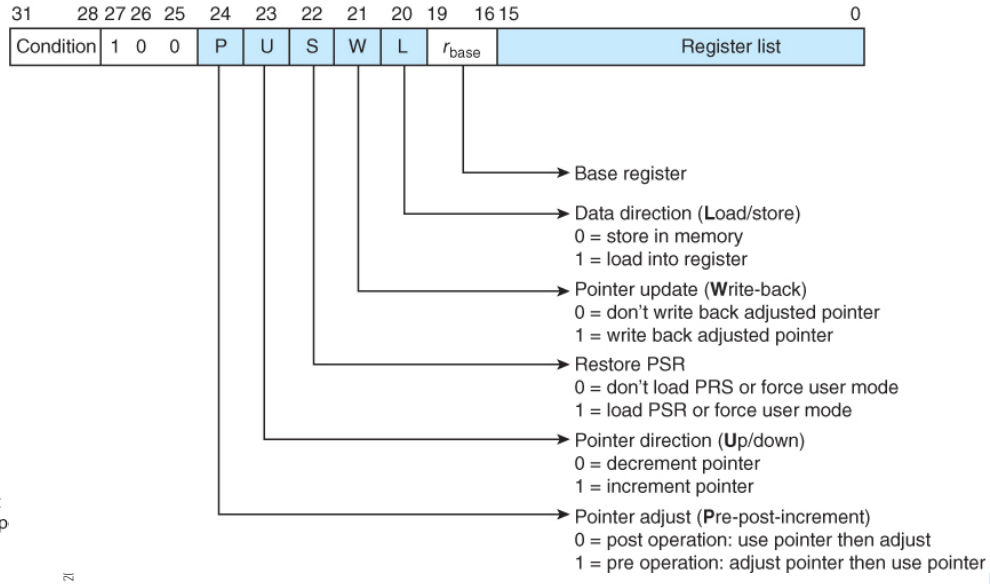


FIGURE 3.58 Encoding ARM's block move instructions



Conversion Tables

$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$
$2^8 = 256$
$2^9 = 512$
$2^{10} = 1024$ (Kilo)
$2^{11} = 2048$
$2^{12} = 4096$
$2^{13} = 8192$
$2^{14} = 16384$
$2^{15} = 32768$
$2^{16} = 65536$
$2^{17} = 131072$
$2^{18} = 262144$
$2^{19} = 524288$
$2^{20} = 1048576$ (Mega)

$(0)_{16} = (0)_{10} = (0000)_2$
$(1)_{16} = (1)_{10} = (0001)_2$
$(2)_{16} = (2)_{10} = (0010)_2$
$(3)_{16} = (3)_{10} = (0011)_2$
$(4)_{16} = (4)_{10} = (0100)_2$
$(5)_{16} = (5)_{10} = (0101)_2$
$(6)_{16} = (6)_{10} = (0110)_2$
$(7)_{16} = (7)_{10} = (0111)_2$
$(8)_{16} = (8)_{10} = (1000)_2$
$(9)_{16} = (9)_{10} = (1001)_2$
$(A)_{16} = (10)_{10} = (1010)_2$
$(B)_{16} = (11)_{10} = (1011)_2$
$(C)_{16} = (12)_{10} = (1100)_2$
$(D)_{16} = (13)_{10} = (1101)_2$
$(E)_{16} = (14)_{10} = (1110)_2$
$(F)_{16} = (15)_{10} = (1111)_2$

ASCII Table

'0' → 0x30
'1' → 0x31
'2' → 0x32
...
'8' → 0x38
'9' → 0x39
...
'A' → 0x41
'B' → 0x42
'C' → 0x43
'D' → 0x44
'E' → 0x45
'F' → 0x46
...
'X' → 0x58
'Y' → 0x59
'Z' → 0x5A
...
'a' → 0x61
'b' → 0x62
'c' → 0x63
'd' → 0x64
'e' → 0x65
'f' → 0x66
...
'x' → 0x78
'y' → 0x79
'z' → 0x7A